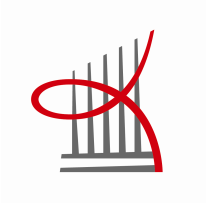


KÄYTTÖJÄRJESTELMÄ AVR-MIKRO-OHJAIMELLE

Janne Koivuranta

Opinnäytetyö
Joulukuu 2011
Tietotekniikka
Sulautetut järjestelmät
Tampereen ammattikorkeakoulu

TAMPEREEN AMMATTIKORKEAKOULU
Tampere University of Applied Sciences

Tekijä: Janne Koivuranta
Työn nimi: Käyttöjärjestelmä AVR-mikro-ohjaimelle
Päivämäärä: 2.12.2011
Sivumäärä: 30
Avainsanat: käyttöjärjestelmä, mikro-ohjain
Koulutusohjelma: Tietotekniikka
Suuntautuminen: Sulautetut järjestelmät

Tiivistelmä

Tämän opinnäytetyön tavoitteena oli suunnitella ja toteuttaa Atmel AVR mikro-ohjaimella toimiva yksinkertainen käyttöjärjestelmä. Käyttöjärjestelmän ominaisuuksiin haluttiin sisällyttää laitteistopohjaisella keskeytyksellä toimiva vuorottaja, joka jakaa suoritinaikaa ajossa oleville ohjelmille. Lisäksi käyttöjärjestelmä tarjoaa tarvittavat palvelut ajossa olevien ohjelmien väliseen synkronointiin. Kaikkien käyttöjärjestelmän palveluiden suoritukseen kuluva aika on ennakoitavissa, joten kyseessä on reaaliaikainen moniajokäyttöjärjestelmä.

Käyttöjärjestelmä toteutettiin kokonaan assemblyllä, eli symbolisella konekielellä. Se kehitettiin GNU/Linux-ympäristössä GNU-projektin assemblerin, debuggerin ja Emacs-tekstieditorin avulla. Käyttöjärjestelmän toimintojen testaamista varten toteutettiin lisäksi C-kielisiä testiohjelmia. Tällä haluttiin havainnollistaa assembly- ja C-kielten yhteiskäyttöä, käyttöjärjestelmää hyödyntävä ohjelmoija voi toteuttaa sovelluksensa valitsemallaan ohjelmointikielellä.

Käyttöjärjestelmään pyrittiin sisällyttämään vain ehdottoman tarpeellisia toimintoja, jotta sen rakenne pysyisi mahdollisimman selkeänä ja lähdekoodin luettavuus hyvänä. Lisätoimintoja, joiden tarpeellisuudesta ei ollut täyttä varmuutta, pyrittiin välttämään, mutta samalla pyrittiin ennakoimaan järjestelmän tuleva laajentaminen lisätoiminnoilla ja eri suoritinarkkitehtuurien tuella. Tavoitteena oli, että kaikkien järjestelmään sisällytettyjen toimintojen olemassaolo on perusteltua.

Opinnäytetyön aihe valittiin, koska haluttiin oppia ymmärtämään käyttöjärjestelmän toiminta ja myös käytännön toteutus alusta lähtien. Samalla haluttiin oppia hieman monimutkaisemman ohjelman toteuttaminen puhtaasti assembly-kielisenä sekä assembly- ja C-kielen yhteiskäyttö samassa projektissa.

Writer:	Janne Koivuranta
Thesis:	Operating System for AVR Microcontroller
Date:	2.12.2011
Pages:	30
Keywords:	operating system, microcontroller
Education program:	Information Technology
Line:	Embedded Systems

Abstract

The goal of this thesis was to design and implement a simple operating system for Atmel AVR microcontroller. The plan was to implement a pre-emptive scheduler based on hardware interruption services of the AVR microcontroller, that would share processing time to tasks running on the operating system. Basic synchronization primitives like signals and semaphores were also desired features. It is possible to predict deadlines of every service that operating system provides, therefore it is a multitasking real-time operating system.

The operating system was written completely in assembly language. It was designed in a GNU/Linux environment using assembler, debugger and Emacs text editor of GNU project. Small test applications were also written in C programming language. The reason for using C was to demonstrate using combination of assembly and C languages together in same project, developer using the operating system may implement his/her applications in higher level language if they seem more suitable for the task.

Only necessary features were incorporated in first version of the operating system, so it's structure would be as clear as possible and readability of the source code would not suffer from overly complicated design. Questionable features were avoided, but it should be easy to add more features and support for other processor architectures during future development of the operating system.

The reason for this kind of thesis subject was to learn operation of operating system in theory and practice. Second goal was to learn an implementation of moderately complex application written completely in assembly language.

ALKUSANAT

Tämä työ on tehty Tampereen ammattikorkeakoulun sulautettujen järjestelmien suuntautumisvaihtoehdon opinnäytetyönä.

Työ on tehty itsenäisesti ja motivaatio sen tekemiseen on ollut kiinnostus ohjelmointia ja käyttöjärjestelmiä kohtaan. Ajatus oman käyttöjärjestelmän toteuttamisesta on alunperin syntynyt sulautettujen järjestelmien opintoihin sisältyvillä matalan tason ohjelmoinnin kursseilla.

Tampereella 2. joulukuuta 2011

Janne Koivuranta

Sisältö

TERMIT JA LYHENTEET

AVR	Atmelin valmistama mikro-ohjainperhe, sisältää sekä 8- että 32-bittisiä ohjaimia.
Rekisteri	Suorittimen sisäinen nopea tiedon tallennuspaikka.
Vuorottaja	(scheduler) taskien suoritusta ohjaava käyttöjärjestelmän ydinkomponentti.
Osoiteavaruus	Alue, joka määrittää joukon toisistaan erillisiä muistiosoitteita tai tiedon tallennuspaikkoja.
SRAM	Static Random-Access Memory, dynaamista ram-muistia nopeampi mutta huomattavasti kalliimpi muistityyppi.
EEPROM	Electrically Erasable Programmable Read-Only Memory, sähköisesti uudelleenohjelmoitava ROM-muisti. Mahdollista uudelleenkirjoittaa tavu kerrallaan.
Flash	EEPROM:n pohjalta kehitetty muistityyppi, yksittäisten tavujen kirjoittaminen ei mahdollista.
Itseään muokkaava ohjelma	(self-modifying code), ohjelma joka muokkaa omia käskyjään suoritusaikaisesti.

1 JOHDANTO

Elektronisten laitteiden yleistyminen ja hinnan putoaminen on johtanut siihen, että lähes kaikissa arkipäiväisissä laitteissa on nykyään pieni laitetta ohjaava tietokone. Sulautetuille järjestelmille on tyypillistä, että käyttäjän ei tarvitse olla tietoinen laitetta ohjaavan tietokoneen olemassaolosta tai toiminnasta.

Laitteen sisään sulautettuja tietokonejärjestelmiä on esimerkiksi televisioissa, puhelimissa, jopa pesukoneissa. Monet sellaisetkin laitteet, joiden toteuttamiseen vaadittiin aiemmin käyttökohteeseen erityisesti suunniteltua elektroniikkaa, voidaan usein toteuttaa yleiskäyttöisillä ohjelmoitavilla mikro-ohjaimilla.

Tällaisissa järjestelmissä on usein erittäin tiukkoja ajoitusvaatimuksia. Esimerkiksi auton moottorinohjauksessa pitkiä viiveitä ei voida sallia, koska palamistapahtuma moottorin sisällä tapahtuu hyvin nopeasti. Palamistapahtumakin on nykyaikaisten suorittimien mittapuulla melko pitkäkestoinen, mutta yleiskäyttöiset käyttöjärjestelmät ovat silti vasteajoiltaan aivan liian hitaita tällaisiin sovelluksiin.

Aikakriittisten prosessien ohjaamiseen tarvitaan niin sanottu reaaliaikakäyttöjärjestelmä (real-time operating system, RTOS). Reaaliaikakäyttöjärjestelmälle on ominaista se, että kaikkien järjestelmän toimintojen suurin mahdollinen viive on tarkkaan määritelty. On olemassa ns. kovia ja pehmeitä reaaliaikakäyttöjärjestelmiä. Pehmeissä reaaliaikakäyttöjärjestelmissä vasteajat yleensä saavutetaan, mutta siitä ei ole täyttä varmuutta. Kovien reaaliaikakäyttöjärjestelmien käyttäytyminen taas on tarkemmin ennustettavaa,
joten aikavaatimuksissa pysyminen voitaan taata.

Reaaliaikakäyttöjärjestelmä ei yleensä ole yleiskäyttöisiin järjestelmiin verrattuna kovin tehokas, jos tehokkuutta mitataan tietyssä ajassa toteutuvien laskutoimitusten tai muiden operaatioiden määrällä. Aikavaatimuksissa pysyminen on ratkaisevaa, ei niinkään puhdas laskentateho.

Tässä työssä suunniteltiin ja toteutettiin yksinkertainen käyttöjärjestelmä, joka voidaan luokitella reaaliaikaiseksi. Aikavaatimuksia ei kuitenkaan ole erikseen määritelty tai mitattu.

2 KÄYTTÖJÄRJESTELMÄN PERUSRAKENNE

Tässä kappaleessa esitellään käyttöjärjestelmän perusrakenne ja käyttö. Käyttöjärjestelmän toiminnan ohjaamiseen on käytettävissä 15 kappaletta järjestelmäkutsuja. Järjestelmäkutsut on toteutettu siten, että järjestelmää voidaan käyttää helposti sekä assembly- että C-kielellä toteutetuissa ohjelmissa.

2.1 Taski

Prosessi on käyttöjärjestelmän perusyksikkö, joka mahdollistaa ohjelman suorittamisen. Käyttöjärjestelmästä riippuen prosessit voivat sisältää erilaista tietoa kuten prosessin omaa tilaa kuvaavia muuttujia, tunnisteen jonka avulla prosessi voidaan erottaa muista prosesseista, tietoja prosessin käyttöönsä varaamista muistialueista sekä tiedostoista ja niin edelleen. Prosessi sisältää myös kopion suorittimen keskeisistä rekistereistä sekä oman prosessikohtaisen pinon, johon sijoitetaan ohjelman varaamat paikalliset muuttujat.

Prosessi on siis yksittäinen suorituskelpoisen ohjelman ilmentymä, jonka avulla käyttöjärjestelmä voi suorittaa ohjelmaa. Samaa ohjelmaa voidaan suorittaa samanaikaisesti monessa erillisessä prosessissa, tällöin ohjelman varaamat paikalliset muuttujat ovat prosessikohtaisia eli prosessit ovat täysin toisistaan erillisiä, vaikka ne suorittavatkin samaa ohjelmakoodia.

Prosesseja kutsutaan pienissä sulautetuille järjestelmille tarkoitetuissa käyttöjärjestelmissä usein taskeiksi tai tehtäviksi, kun taas esimerkiksi Unix-tyyppisissä järjestelmissä käytetään prosessi-nimitystä. Monimutkaisemmissa käyttöjärjestelmissä prosesseilla on myös ominaisuuksia, joita ei tässä järjestelmässä ole, kuten prosessikohtainen täysin erillinen osoiteavaruus ja muita suojausmekanismeja.

On tärkeää huomata, että säikeistystä tukevissa käyttöjärjestelmissä yksittäinen prosessi voi suorittaa samanaikaisesti useita erillisiä taskeja, joilla on kuitenkin yhteisiä muistialueita, tiedostoja ja muita resursseja. Taskia voidaankin pitää prosesseja, säikeitä ja muita suorituskelpoisia kokonaisuuksia kuvaavana yleiskäsitteenä ja taskia pidetään usein prosessin synonyyminä.

Nykyaikaisissa moniajokäyttöjärjestelmissä prosessi siis kuvaa suoritettavan ohjelman ilmentymää, mutta ohjelman osia voidaan suorittaa rinnakkaisina taskeina säikeiden avulla. Suoritettavan ohjelmakoodin vaihtamista toiseen kutsutaan taskin vaihdoksi riippumatta siitä, vaihtuuko prosessi kokonaan toiseen, vai vaihdetaanko suoritettavaksi toinen saman prosessin sisältämä säie.

Tässä käyttöjärjestelmässä sekä itse järjestelmä että sitä hyödyntävät ohjelmat sijaitsevat samassa suorittimen osoiteavaruudessa ja ne myös linkitetään kääntämisen jälkeen yhdeksi mikro-ohjaimen ohjelmamuistiin tallennettavaksi binäärimuotoiseksi tiedostoksi. Käyttöjärjestelmä ja sen alaisuudessa suoritettavat ohjelmat ovat siis käytännössä saman ohjelman rinnakkain suoritettavia taskeja. Järjestelmän suorittamia ohjelmia kutsutaankin siis juuri taskeiksi.

2.2 AVR-mikro-ohjaimen rakenne

Seuraavassa kappaleessa esitellään taskin tietorakenne lähdekooditasolla, eli miten taski tallennetaan AVR-mikro-ohjaimen muistiin. Ennen kuin taskin rakennetta aletaan tarkastelemaan, on tärkeää käydä läpi siihen olennaisesti liittyvät seikat AVR-mikro-ohjaimen rakenteesta. Suorittimen käskykantaa kokonaisuudessaan ei käsitellä, mutta yksittäisiä käskyjä mainitaan.

2.2.1 Rekisterit

AVR-mikro-ohjaimen suoritin sisältää 32 kappaletta yleiskäyttöisiä työrekistereitä, eli suorittimen sisäisiä nopeasti toimivia muistipaikkoja. Kaikki suorittimen ALU:n toteuttamat operaatiot kuten yhteen- ja vähennyslaskut, loogiset operaatiot, ym. toiminnot tapahtuvat työrekistereiden avulla.

Useimmat suorittimen käskyistä voivat hyödyntää suoraan kaikkia yleiskäyttöisiä rekistereitä, mutta 16 ensimmäiseen rekisteriin eli rekistereihin R0 - R15 liittyy kuitenkin joitain rajoituksia. Niihin ei esimerkiksi voida suoraan ladata välitöntä (ohjelmakoodiin sisältyvää) tietoa.

Rekisterit R26 - R31 muodostavat kolme rekisteriparia, joita voidaan käyttää 16-bittisinä osoittimina. Niitä voidaan käyttää tiedon tallentamiseen ja lukemiseen muistista ohjelmallisesti laskettujen epäsuorien osoitteiden avulla. Osoittimia kutsutaan X-, Y- ja Z-osoittimiksi.

Tilarekisteri SREG on 8-bittinen rekisteri, joka sisältää tärkeää tietoa suorittimen tilasta. Tilarekisteriin tallentuvat esimerkiksi vertailuoperaatioiden tulokset, matemaattisen operaation mahdollinen muistibitti, vähennyslaskun tuloksen etumerkki, jne. Tilarekisteri sisältää myös laitteistokeskeytysten sallintaa merkitsevän bitin ja yhden yleiskäyttöisen väliaikaiseen tiedon tallentamiseen varatun bitin.

Ohjelmalaskuri on suorittimen sisäinen rekisteri, joka osoittaa aina ohjelmamuistissa sijaitsevaan seuraavaan suoritettavaan käskyyn. Ohjelmamuistin osoitus tapahtuu sanamuotoisesti (word-addressing), eli ohjelmalaskurilla valitaan 16-bitin kokoinen lohko ohjelmamuistista. Kun käsky suoritetaan, ohjelmalaskurin arvo kasvaa automaattisesti käskyn pituuden verran ja siirtyy osoittamaan seuraavaa käskyä.

Myös ohjelman suorituksen ohjaaminen tapahtuu ohjelmalaskurin avulla, ehdolliset ja ehdottomat hyppykäskyt kuten *jmp* ja *breq* muuttavat siis ohjelmalaskurin arvoa. Aliohjelmaa kutsuttaessa aliohjelman osoite kopioidaan ohjelmalaskuriin, josta suoritin lukee aliohjelman osoitteen. Ohjelmalaskurin arvo voidaan lukea myös työrekistereihin kuuluvasta Z-osoittimesta epäsuoralla hyppykäskyllä *ijmp* tai epäsuoralla aliohjelmakutsulla *icall*.

Käskyjen pituus ei ole vakio, vaan ne kuluttavat 2 tai 4 tavua ohjelmamuistia. Esimerkiksi käsky *subi*, joka vähentää kohderekisteristä välittömän arvon, on kahden tavun pituinen. Käskyn ensimmäinen tavu sisältää itse käskyn sekä kohderekisterin numeron ja toinen tavu vähennettävän arvon. Käskyn suorittaminen kasvattaa siis ohjelmalaskurin arvoa yhdellä.

Pino-osoitin on rekisterien *SPL* ja *SPH* muodostama 16-bittinen rekisteripari, joka osoittaa pinon seuraavaa vapaata kirjoituspaikkaa. Pino kasvaa alhaalta ylöspäin, eli pino-osoittimen arvo pienenee yhdellä aina kun pinoon tallennetaan yksi tavu.

Aliohjelmaa kutsuttaessa tai keskeytyspalveluohjelman käynnistyessä paluuosoite, eli kutsua seuraavan käskyn osoite kopioidaan automaattisesti pinoon. Ali- tai keskeytyspalveluohjelmasta palattaessa paluuosoite luetaan pinosta käskyllä *ret* tai *reti* ja ohjelman suoritus jatkuu aiemmin tallennetusta osoitteesta. Minkä tahansa työrekisterin arvo voidaan kirjoittaa pinoon käskyllä *push* ja lukea sieltä käskyllä *pop*.

Kaikki edellämainitut rekisterit ovat mikro-ohjaimen suorittimen sisäisiä rekistereitä. Ohjain sisältää kuitenkin suorittimen lisäksi myös oheislaitteita kuten neljä 8-bittistä I/O-porttia, ajastimia ja A/D-muuntimen. Oheislaitteiden ohjaamista varten on käytettävissä 64 kappaletta I/O-rekistereitä, jotka on sijoitettu suorittimen käyttömuistiin heti työrekistereiden jälkeen. Myös tilarekisteri ja pino-osoitin sijaitsevat I/O-rekisterien alueella sen kolmessa viimeisessä tavussa.

Ohjelmalaskuria ja pino-osoitinta lukuunottamatta I/O-rekisterit eivät sisälly taskin kontekstiin, joten niitä ei käsitellä tarkemmin. Käyttöjärjestelmä hyödyntää kuitenkin joitakin rekistereistä vuorottajaa ohjaavan vertailukeskeytyksen alustuksessa.

2.2.2 Muisti

AVR on modifioituun Harvard-arkkitehtuuriin perustuva mikro-ohjain, eli ohjelma- ja käyttömuisti ovat toisistaan erotettuja ja sijaitsevat erillisissä osoiteavaruuksissa. ATMega32 sisältää 2 kilotavua SRAM-käyttömuistia, 32 kilotavua flash-ohjelmamuistia sekä yhden kilotavun EEPROM-muistia ohjelman asetusten ja muiden tietojen pysyvään tallentamiseen. EEPROM-muistia ei hyödynnetä tässä käyttöjärjestelmässä, mutta sen käsittely tapahtuu tarkoitukseen varattujen I/O-rekisterien avulla. Myös EEPROM-muisti sijaitsee erillisessä osoiteavaruudessa.

Ohjelmamuistia on mahdollista lukea ja kirjoittaa ohjelmallisesti käskyillä *lpm* ja *spm*, ohjelmamuistia voidaan siis käsitellä kuten käyttömuistia. Käyttöjärjestelmä ei tällä hetkellä hyödynnä kyseisiä käskyjä, mutta niiden avulla olisi mahdollista lukea ja suorittaa ohjelmia ohjaimeen kytketystä ulkoisesta muistista, kuten muistitikulta. Niiden avulla on myös mahdollista toteuttaa itseään muokkaavia ohjelmia.

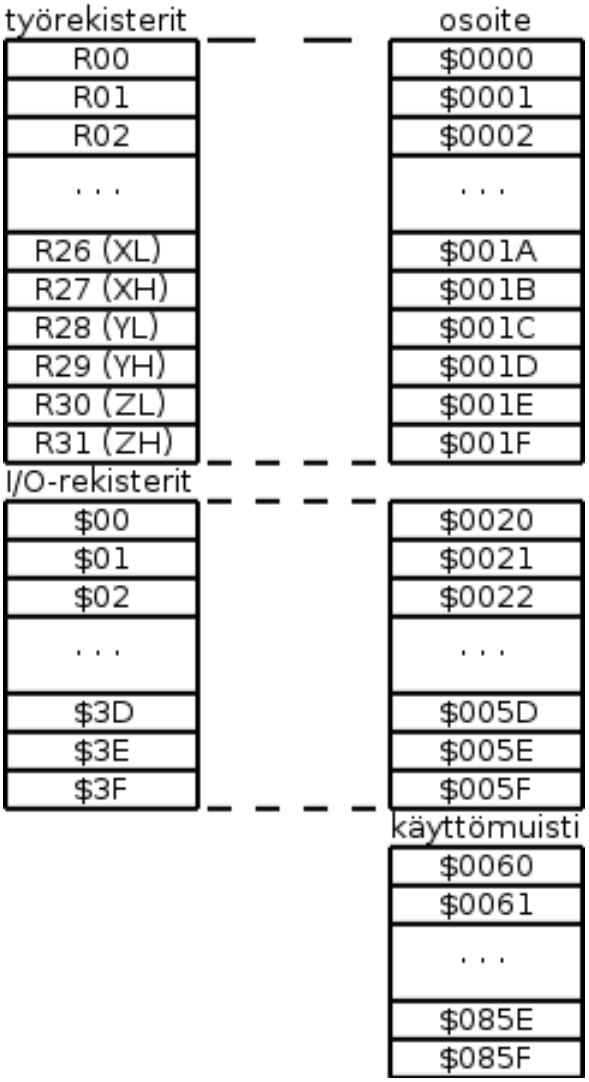
C-kääntäjä varaa ohjelman tietosegmentissä (data segment) sijaitseville muuttujille tilaa sekä ohjelma- että käyttömuistista. Ohjelman käynnistysvaiheessa käyttömuistissa sijaitsevat muuttujat alustetaan ohjelmamuistissa sijaitsevilla alkuarvoilla.

AVR-suorittimessa sekä käyttömuisti että rekisterit sijaitsevat samassa osoiteavaruudessa. Rekistereiden käsittelyyn voidaan siis käyttää myös muistin käsittelyyn varattuja käskyjä, mutta työ- ja I/O-rekisterien käsittelyyn varatut käskyt toimivat vain sillä osoitealueella, missä ko. rekisterit sijaitsevat.

Kuvassa 2.1 esitellään rekistereiden ja käyttömuistin sijainti suorittimen osoiteavaruudessa. Työrekisterit sijaitsevat siis 32 ensimmäisessä osoitteessa. Heti niiden jälkeen osoiteavaruudessa sijaitsevat 64 I/O-rekisteriä ja I/O-rekistereiden jälkeen 2 kilotavua käyttömuistia.

Jotta taskit eivät häiritsisi toisiaan, on niiden suoritusympäristö eli niin sanottu konteksti tallennettava aina, kun taskin suoritus päättyy. Seuraavan taskin suorituksen alkaessa on puolestaan kyseisen taskin aiemmin tallennettu konteksti palautettava, jotta suoritin palautuu samaan tilaan, missä se oli taskin suorituksen päättyessä.

Konteksti sisältää kopion suorittimen kaikista työrekistereistä, tilarekisteristä, ohjelmalaskurista ja pino-osoittimesta. Jokaisella taskilla on oma erillinen pino, minne työrekisterit, tilarekisteri ja ohjelmalaskuri tallennetaan. Ainoastaan pino-osoitin sijaitsee pinosta erillisessä kiinteässä muistipaikassa.

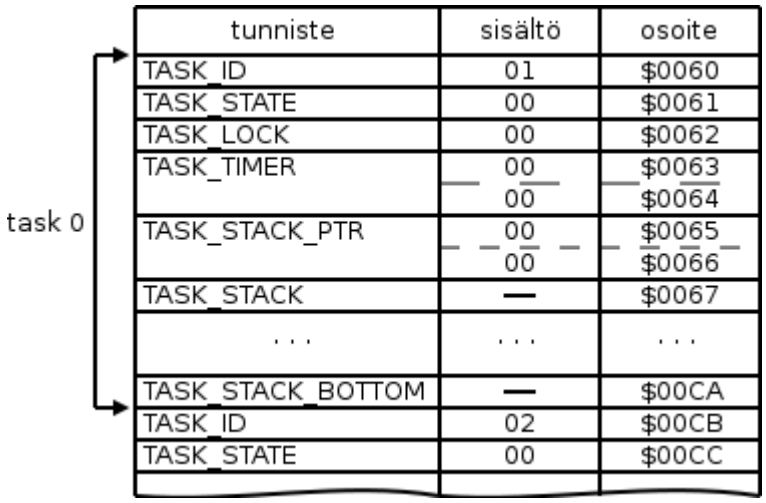


Kuva 2.1: Rekisterit ja käyttömuisti

2.3 Taskin tietorakenne

Taskien tietorakenteet sijaitsevat peräkkäin heti käyttömuistin alussa, ensimmäisen tietorakenteen osoite heksadesimaalimuodossa on siis *0060*. Tietorakenne koostuu kahdesta tilamuuttujasta, tunnisteesta, yhdestä ajastinlaskurista, pino-osoittimesta, sekä pinosta, jonka koko on mahdollista määritellä käännösvaiheessa. Taskien suurin samanaikainen määrä on myös käyttäjän määriteltävissä, jokainen taski kuluttaa aina saman verran käyttömuistia riippumatta siitä, onko taski juuri sillä hetkellä käytössä. Tietorakenne muistuttaa C-kielen *struct*-tietorakennetta.

Kuvassa 2.2 on esitelty taskin tietorakenne ja sisältö heti käyttöjärjestelmän käynnistyksen jälkeen, pinon koko on 100 tavua. Kahden tavun kokoisissa muuttujissa (pino-osoitin ja ajastin) eniten merkitsevä tavu sijaitsee aina vähiten merkitsevää tavua suuremmassa osoitteessa. Pinon muistialuetta ei erikseen alusteta millään tietyllä alkuarvolla, joten sen sisältö on merkitty viivalla.



Kuva 2.2: Taskin tietorakenne muistissa

Kuvan 2.2 tunnistesarakkeessa olevat nimet ovat käyttöjärjestelmän lähdekoodin numeerisia vakioita, jotka kertovat kyseisen tietoalkion osoitteen suhteessa taskin alkuosoitteeseen. Esimerkiksi *TASK_ID* vastaa arvoa nolla, koska tunniste sijaitsee heti taskin ensimmäisessä tietoalkiossa. Pino-osoitin taas sijaitsee viiden tavun päässä taskin alkuosoitteesta, joten vakio *TASK_STACK_PTR* on arvoltaan viisi.

TASK_ID on yhden tavun kokoinen tunniste. Taskien tunnisteet asetetaan käyttöjärjestelmän alustusvaiheessa, ja ne ovat aina nollaa suurempia kokonaislukuja.

TASK_STATE on taskin tilaa kuvaava tilamuuttuja, jonka arvo voi olla jokin vakioista *TASK_KILLED*, *TASK_RUNNING*, *TASK_STOPPED* ja *TASK_SLEEPING*. Se kertoo, onko tietyssä muistiosoitteessa sijaitseva taski olemassa, eli onko sitä vielä edes luotu tai onko sen suoritus lopetettu.

TASK_STATE kertoo myös, onko taski ajovalmis vai odottaako se signaalia, muteksia, semaforia tai ajastinta tai onko sen suoritus pysäytetty signaalilla toisesta taskista käsin. Tilassa *TASK_KILLED* olevaa taskia ei ole käyttäjän näkökulmasta katsottuna olemassa, eli sitä ei voida suorittaa, vaikka taskeille onkin aina varattuna kiinteä määrä tilaa.

TASK_LOCK kertoo, onko taskin suoritus lukittu muteksilla tai semaforilla. Muuttujaan sijoitetaan muteksin tai semaforin numero sekä tunnistebitti, joka kertoo, onko lukitus tyypiltään muteksi vai semafori. Arvo on aina nolla, jos taskia ei ole lukittu.

TASK_TIMER on kahden tavun kokoinen ajastinlaskuri. Järjestelmäkutsu *sleep* pysäyttää taskin suorituksen ja sijoittaa argumenttina annetun arvon *TASK_TIMER* muuttujaan. Järjestelmä pienentää muuttujan arvoa yhdellä aina kun vuorottajaa kutsutaan laitteistokeskeytyksellä. Taskin suoritus jatkuu, kun muuttujan arvo pienenee nollaan.

TASK_STACK_PTR on kahden tavun kokoinen pino-osoitin, joka osoittaa aina jotain muistipaikkaa taskin omassa pinossa. Suoritettavan taskin vaihtuessa toiseen suorittimen ”oikea” pino-osoitin kopioidaan talteen taskin omaan pino-osoittimeen. Osoittimen arvo luetaan ja sijoitetaan suorittimen pino-osoittimeen, kun taskin suoritus jatkuu.

TASK_STACK ja *TASK_STACK_BOTTOM* osoittavat pinon ensimmäistä ja viimeistä tavua.

2.4 Taskin luominen

Uusia taskeja voidaan luoda järjestelmäkutsulla *create*. Kutsulle annetaan parametrina taskin suorittaman ohjelman osoite ja se palauttaa taskille annetun tunnisteen. Tunnistetta ei siis voida itse valita, vaan järjestelmä valitsee sen automaattisesti.

Taski voidaan luoda vain, jos jo olemassaolevien taskien määrä on samanaikaisten taskien suurinta mahdollista määrää pienempi. Järjestelmäkutsu siis onnistuu, jos taskien suurin määrä on viisi ja kutsun aikana taskeja on käytössä neljä. Jos kaikki taskit on jo varattu, kutsu epäonnistuu ja palauttaa arvon *RET_NOAVAIL*.

Kun uutta taskia luodaan, on ensin laskettava pinon pinnan eli seuraavan vapaan kirjoituspaikan osoite. Se lasketaan vähentämällä pinon pohjan eli viimeisen tavun osoitteesta kontekstin koko. Pinon pinta sijaitsee siis osoitteessa *TASK_STACK_BOTTOM - TASK_CONTEXT_SIZE*. Kontekstiin sisältyy 32 kappaletta työrekistereitä, tilarekisteri ja kahden tavun kokoinen ohjelmalaskuri, joten vakion *TASK_CONTEXT_SIZE* arvo on 35.

Kun pinnan osoite on laskettu, se tallennetaan taskin pino-osoittimeen. Pino-osoitin on kahden tavun kokoinen ja osoitteen eniten merkitsevä tavu tallennetaan suurempaan osoitteeseen.

Kun pino-osoitin on tallennettu, pinoon tallennetaan ohjelmalaskuri eli järjestelmäkutsulle annettu argumentti, joka kertoo taskin suorittaman ohjelman alkuosoitteen. Ohjelmalaskuria kutsutaan tässä yhteydessä paluuosoitteeksi, koska taskiin siirtyminen tapahtuu paluukäskyllä *reti*, joka aktivoi laitteistokeskeytykset ja hyppää osoitteeseen, joka luetaan suorittimen pino-osoittimen osoittamasta paikasta.

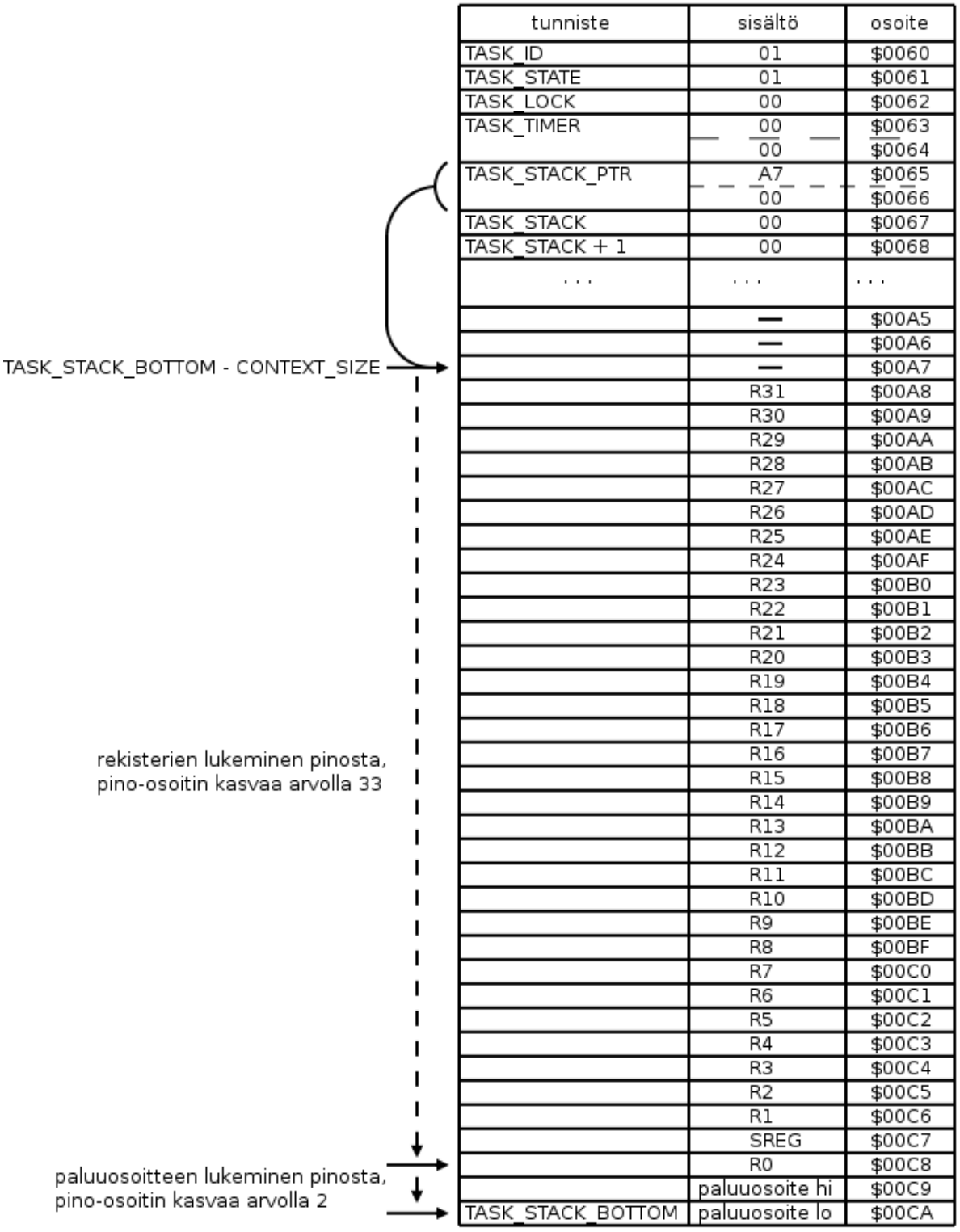
Paluuosoite tallennetaan pinon pohjalle, eli sen vähemmän merkitsevä tavu tallennetaan osoitteeseen *TASK_STACK_BOTTOM*. Pino kasvaa alhaalta ylöspäin, eli pino-osoittimen arvo pienenee aina, kun pinoon sijoitetaan tietoa. Tästä syystä paluuosoittimen vähiten merkitsevä tavu tallennetaan suurempaan osoitteeseen pinon pohjalle.

Kontekstin koko on 35 tavua ja paluuosoitteen 2 tavua, joten paluuosoitteen ja pinon pinnan väliin jää 33 tavun verran tyhjää tilaa. Tähän tilaan tallennetaan työrekisterit ja tilarekisteri. Kun taski on luotu, pinon rekistereille varattuun tilaan ei ole vielä tallennettu mitään, koska taskia ei olla vielä suoritettu kertaakaan. Rekistereille on kuitenkin varattava tilaa, koska taskinvaihdon yhteydessä suorittimen rekistereihin ladataan pinossa olevat arvot. Taskin ensimmäisellä ajovuorolla rekistereihin siis ladataan ne arvot, joita sen pinoon kuuluvalla muistialueella sattuu sillä hetkellä olemaan, siitä ei kuitenkaan ole mitään haittaa.

Kun luotu taski siirtyy ensimmäisen kerran ajovuoroon, rekisterien arvot ladataan pinosta ja pino-osoitin siirtyy osoittamaan pohjalla sijaitsevaa pino-osoitetta (sen eniten merkitsevää eli osoitteeltaan pienempää tavua). Tämän jälkeen vuorottaja suorittaa käskyn *reti*, joka lukee paluuosoitteen ja hyppää taskin ohjelmaan. Pino-osoittimen arvo kasvaa tällöin kahdella tavulla osoittamaan pinon pohjaa, eli pino on paluukäskyn jälkeen tyhjä.

Jos taskin ohjelma on kirjoitettu C-kielellä, kääntäjä sijoittaa ohjelman varaamat paikalliset muuttujat pinoon. Ne siis säilyvät pinossa taskinvaihtojen välillä ja ovat aina taskikohtaisia, vaikka monet taskit suorittaisivat samaa ohjelmaa. Taskia suoritetaan, kunnes laitteistopohjainen vertailukeskeytys keskeyttää sen ja suoritus siirtyy käyttöjärjestelmän vuorottajaan. Tällöin sen hetkinen ohjelmalaskuri eli paluuosoite tallennetaan taas pinoon ja taskin suoritus jatkuu myöhemmin samasta kohdasta.

Kuvasta 2.3 on nähtävissä taskin muistialueen sisältö heti taskin luomisen jälkeen. Rekisterien sijainti pinossa on merkitty kuvaan selkeyden vuoksi.



Kuva 2.3: Uuden taskin tietorakenteen sisältö

2.5 Käyttöjärjestelmän alustaminen

Käyttöjärjestelmän alustaminen ja käyttöönotto tapahtuu järjestelmäkutsulla *init*. Kutsulle annetaan argumenttina ensimmäisen niin sanotun päätaskin osoite, se luo ensimmäisen taskin ja alkaa välittömästi suorittaa sitä. Päätaskin tunniste on aina 1.

Käyttöjärjestelmä siis alustetaan pääohjelmasta käsin (C-kielen tapauksessa main-funktio) ja kutsun jälkeen pääohjelmaan ei ole enää mahdollista palata. Pääohjelma toimii siis vain järjestelmän käynnistyslataajana ja sen jälkeen kaikkien ohjelmien suoritus tapahtuu järjestelmän taskien avulla. Kutsusta ei siis koskaan palata, joten sillä ei ole myöskään paluuarvoa.

Taskien suurin mahdollinen määrä asetetaan vakion *TASK_CNT* avulla. Päätaski varaa yhden muistialueen itselleen, joten jos *TASK_CNT:n* arvo on viisi, voidaan kutsulla *create* luoda vielä neljä taskia. Arvon on siis oltava aina nollaa suurempi kokonaisluku. Taskeja on myös mahdollista tuhota, jolloin taskille varattu tila vapautuu ja se voidaan luoda uudestaan. Taski voi tuhota itsensä järjestelmäkutsulla *exit*.

Alustus varaa tilaa myös yhdelle ylimääräiselle taskille, jota ei lasketa mukaan vakion *TASK_CNT* ilmoittamaan taskien määrään. Taski on niin sanottu idletaski, jota suoritetaan silloin, kun mitään muuta taskia ei voida suorittaa. Tällainen tilanne syntyy esimerkiksi silloin, kun kaikki taskit ovat pysäyttäneet itsensä määräajaksi järjestelmäkutsulla *sleep*.

Idletaski sijaitsee käyttöjärjestelmän sisällä, eikä käyttäjä voi muuttaa sen sisältöä. Käyttöjärjestelmän nykyisessä versiossa idletaski suorittaa päättymätöntä silmukkaa, eikä tee mitään muuta. Sen sisällä olisi kuitenkin mahdollista asettaa suoritin virransäästötilaan tai esimerkiksi järjestellä dynaamiseen muistinvaraukseen liittyviä tietorakenteita. Kyseisiä toimintoja ei ole kuitenkaan vielä toteutettu. Idletaski vie saman verran käyttömuistia kuin muutkin taskit, koska sen pino on saman kokoinen. Tämä tullaan myöhemmin muuttamaan.

Käyttöjärjestelmän alustusta ohjaavat vakiot *CPU_FREQ*, *SCHED_FREQ*, *TASK_CNT*, *STACK_SIZE*, *SEM_CNT* ja *MTX_CNT*. Jos vakioita muutetaan, on käyttöjärjestelmä käännettävä uudestaan, jotta muutokset tulisivat voimaan.

2.6 Vuorottaja

Tässä kappaleessa esitellään käyttöjärjestelmän ydinkomponentti vuorottaja, joka huolehtii taskinvaihdosta ja suoritinajan jakamisesta taskeille. Vuorottaja voi olla yhteistyöhön perustuva (co-operative), jolloin taskin on erikseen luovutettava ajovuoro seuraavalle taskille, tai keskeyttävä, jolloin käyttöjärjestelmä voi tarvittaessa keskeyttää minkä tahansa taskin suorituksen, vaikka se suorittaisi päättymätöntä silmukkaa.

2.6.1 Vertailukeskeytys

Tässä käyttöjärjestelmässä vuorottajan toiminta perustuu laitteistopohjaiseen vertailukeskeytykseen. Vertailukeskeytys tapahtuu tasaisin väliajoin, ja se käynnistää vuorottajan automaattisesti riippumatta sillä hetkellä suoritettavan taskin tilasta. Vuorottaja tallentaa taskin kontekstin ja etsii seuraavan ajokelpoisen taskin. Jos ajokelpoista taskia ei löydy, ajovuoro siirtyy takaisin keskeytetylle taskille. Jos yhtäkään ajokelpoista taskia ei löydy, järjestelmä siirtyy suorittamaan erityistä järjestelmän sisäistä idletaskia.

Vuorottajaa ohjataan mikro-ohjaimen 8-bittisellä ajastimella ja siihen liittyvällä vertailukeskeytyksellä. Ajastimen arvo kasvaa suorittimen kellokiteen ohjaamana ja sen toiminta on suorittimesta riippumatonta. Järjestelmää alustettaessa ajastin asetetaan toimimaan jakajalla, jonka arvo on 256, ajastimen arvo siis kasvaa yhdellä jokaista 256:ta suoritinta ohjaavaa kellopulssia kohden. Jos suorittimen kellotaajuus 16 megahertsiä, ajastin toimii 62,5 kilohertsin taajuudella.

Mikro-ohjain vertaa ajastimen arvoa vertailurekisteriin ja suorittaa vertailukeskeytyksen aina kun arvot täsmäävät. Vuorottaja asettaa vertailurekisterille uuden arvon siten, että keskeytykset tapahtuvat aina tasaisin väliajoin. Väliaikaa eli vuorottajan taajuutta voidaan muuttaa vakion *SCHED_FREQ* avulla, taajuus on oletuksena yksi kilohertsi eli vuorottaja suoritetaan tuhat kertaa sekunnissa.

2.6.2 Kiertovuorottelu

Erilaisia vuorottajan toimintaperiaatteita on olemassa useita, ja eräs yksinkertaisimmista on tässäkin järjestelmässä käytetty kiertovuorottelu (round robin) /6, s. 84/. Kiertovuorottelu ei perusmuodossaan sisällä taskikohtaisia prioriteetteja, eli se jakaa aina yhtä paljon suoritinaikaa kaikille taskeille, jokainen taski saa siis käyttöönsä yhtäläisen aikaviipaleen. Kun vuorottaja käynnistyy, se etsii seuraavan suoritusvalmiin taskin ja alkaa suorittaa sitä, suoritusjärjestys siis kiertää ensimmäisestä taskista kohti viimeistä ja siirtyy sitten taas takaisin ensimmäiseen taskiin.

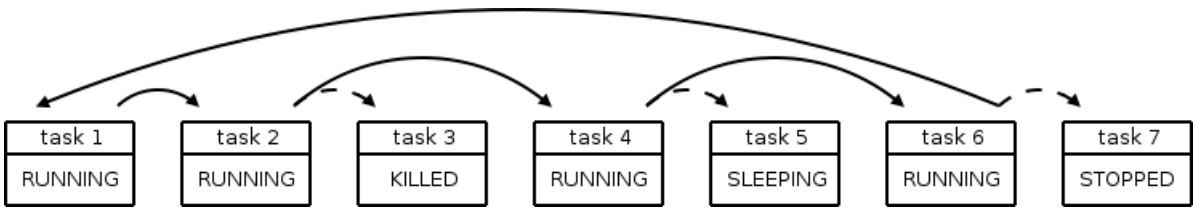
Kiertovuorottelu on yksinkertainen ja varmatoiminen menetelmä. Mikään taski ei voi nälkiintyä (starvation) eli jäädä ilman suoritinaikaa, koska prioriteetteja ei ole, joten korkean prioriteetin taski ei voi estää tai viivästää matalan prioriteetin taskin suoritusta. Toisaalta menetelmä sopii huonosti tilanteisiin, joissa osa taskeista tarvitsisi enemmän suoritinaikaa. Jokainen suoritettava taski vähentää suoraan muille taskeille jäävää suoritinaikaa, joten osa taskeista voi toimia liian hitaasti. Taski voi kuitenkin luovuttaa ajovuoron vapaaehtoisesti seuraavalle taskille saatuaan oman tehtävänsä suoritetuksi.

Toinen kiertovuorottelun heikkous on vasteaikojen pituus. Koska taskeja suoritetaan aina samassa järjestyksessä, voi aikakriittistä tehtävää suorittava taski joutua odottamaan kaikkien muiden jonossa olevien taskien valmistumista, ennen kuin se saa suorittimen omaan käyttöönsä. Ongelma korostuu, jos samanaikaisesti suoritettavien taskien määrä on suuri, ja menetelmä sopii melko huonosti aikakriittisiin reaaliaikajärjestelmiin.

Kiertovuorottelu kuluttaa paljon suoritinaikaa, koska taskinvaihdot tapahtuvat tasaisin väliajoin riippumatta siitä, onko taskinvaihto tarpeellinen. Prioriteetteihin perustuva vuorottaja voi pitää korkean prioriteetin taskin ajovuorossa yhtämittaisesti ja suorittaa taskinvaihdon matalan prioriteetin taskiin vain silloin, kun se on välttämätöntä. Kuvassa 2.4 on esitelty kiertovuorottajan toimintaperiaate. Vuorottaja etsii aina seuraavaa ajovalmista taskia ja suorittaa taskinvaihdon löydettyään taskin, jonka tilamuuttujan arvo on *TASK_RUNNING*.

Vuorottaja huolehtii myös järjestelmäkellosta ja taskien ajastimista. Vuorottaja suoritetaan tasaisin väliajoin, joten myös kello ja ajastimet toimivat aina samalla nopeudella. Koska taskinvaihto voidaan kuitenkin suorittaa myös keskeytysjaksojen välillä esim. taskin tuhoutuessa, vuorottajalle on erillinen järjestelmäkutsu, jota toiset järjestelmäkutsut käyttävät. Taskinvaihto voidaan suorittaa myös järjestelmäkutsulla *schedule*.

Nykyinen ratkaisu on kuitenkin ongelmallinen, koska vertailukeskeytyksen ohjaama vuorottaja saattaa suorittaa taskinvaihdon heti *schedule*-kutsun jälkeen. Parempi ratkaisu olisi varata ajastimien ja järjestelmäkellon käsittelyyn oma vuorottajasta täysin erillinen laitteistokeskeytys tai käyttää vakionopeudella etenevää erillistä reaaliaikakelloa.



Kuva 2.4: Kiertovuorottelu

3 SIGNAALIT

Signaali on mekanismi, jonka avulla taskin suoritusta voidaan ohjata toisesta taskista käsin, se on siis eräs prosessien välisen kommunikaation (inter-process communication, IPC) muoto. Tässä käyttöjärjestelmässä signaalit muistuttavat Unix-järjestelmien vastaavia, mutta ovat huomattavasti yksinkertaisempia ja niiden määrä on pienempi.

Taski voi pysäyttää itsensä ja jäädä odottamaan signaalia järjestelmäkutsulla *sleep*. Kutsulle annetaan argumenttina kahden tavun kokoinen ajastimen lähtöarvo, joka kertoo, kuinka pitkäksi aikaa taski pysähtyy. Kun taskin suoritus jatkuu, kutsu palauttaa arvon *RET_OK*, jos taski käynnistyi uudestaan ajastimella, ja jäljellä olevan ajastimen arvon, jos taski käynnistettiin signaalilla.

Jos argumentti on nollaa suurempi kokonaisluku, käyttöjärjestelmä lähettää taskille käynnistyssignaalin, kun argumentin ilmoittama määrä vuorottajakutsuja on tapahtunut. Jos vakiolla *SCHED_FREQ* ilmoitettu vuorottajan taajuus on tuhat hertsiä, taski pysähtyy sekunniksi, kun argumentti on arvoltaan tuhat. Jos argumentti on nolla, taski voi palata ajovuoroon vain vastaanotettuaan toisen taskin lähettämän signaalin.

Taski voi lähettää signaalin toiselle taskille järjestelmäkutsulla *signal*. Kutsulla on kaksi tavun kokoista parametria: tunniste, joka kertoo, mille taskille signaali lähetetään ja lähetettävän signaalin numero. Mahdollisia signaaleja kuvataan numeerisilla vakioilla, ja ne ovat nimeltään *SIG_STAT*, *SIG_KILL*, *SIG_STOP* ja *SIG_CONT*.

Järjestelmässä ei ole erillisiä kutsuja taskin olemassaolon tai tilan tiedusteluun, eikä myöskään taskin tuhoamiseen. Myös nämä toimenpiteet tapahtuvat *signal*-järjestelmäkutsun ja sen paluuarvon avulla.

Unix-järjestelmistä poiketen käyttöjärjestelmän nykyisessä versiossa ei ole mahdollista estää signaalien saapumista signaalimaskeilla /1/ tai asettaa niille erillisiä signaalinkäsittelijöitä, eli signaalin saapuessa suoritettavia aliohjelmia /2/.

Signaali *SIG_KILL* voidaan lähettää mille tahansa olemassaolevalle taskille, eli aina kun taskin tila ei ole *TASK_KILLED*. Signaali lopettaa taskin suorituksen välittömästi, ja se voidaan käynnistää vain luomalla taski uudestaan järjestelmäkutsulla *create*.

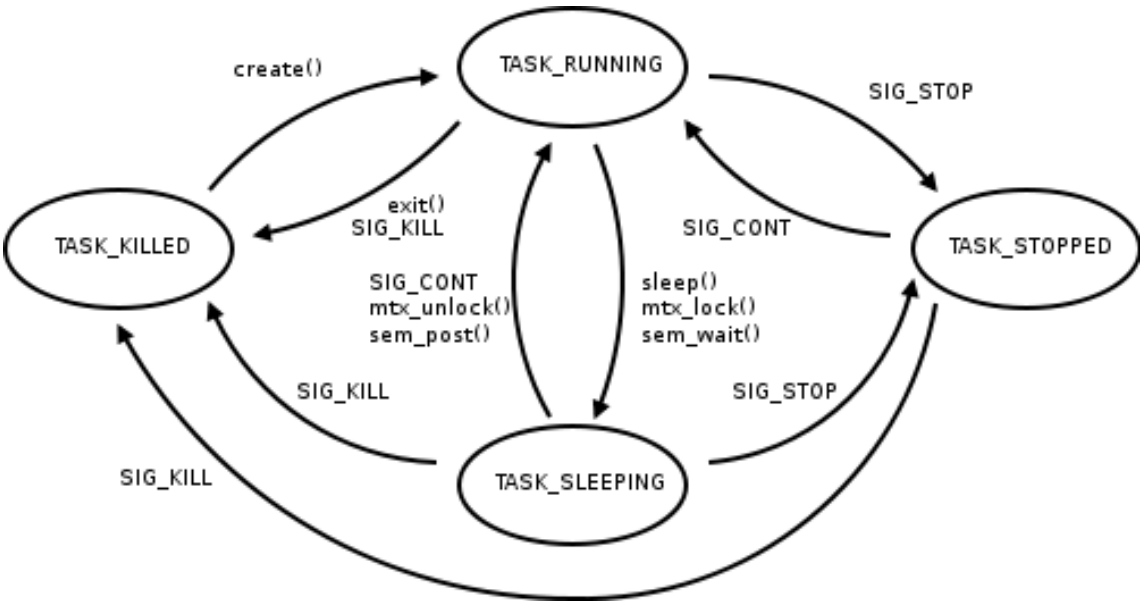
Signaali *SIG_STOP* voidaan myös lähettää mille tahansa taskille. Se lopettaa taskin suorituksen välittömästi, mutta taski voidaan käynnistää uudelleen. Signaalilla ei ole vaikutusta, jos taski on jo aiemmin pysäytetty signaalilla *SIG_STOP*, eli sen tila on *TASK_STOPPED*. Jos taski odottaa muteksia, semaforia tai ajastinta, se siirtyy tilasta *TASK_SLEEPING* tilaan *TASK_STOPPED*.

Signaali *SIG_CONT* siirtää *TASK_STOPPED*- tai *TASK_SLEEPING*-tilassa olevan taskin *TASK_RUNNING*-tilaan, eli taski muuttuu ajovalmiiksi. Taskin suoritus ei jatku välittömästi, vaan vasta, kun vuorottaja käynnistää sen. Jos taski on *TASK_SLEEPING*-tilassa ja se on lukittu muteksilla tai semaforilla, suoritus jatkuu lukituksesta huolimatta.

Kaikkia edellämainittuja signaaleja lähetettäessä järjestelmäkutsu *signal* palauttaa paluuarvon *RET_OK*, jos taski, jolle signaalia yritetään lähettää, on olemassa ja signaalin numero on sallittu. Jos argumenttina annetun tunnisteen perusteella ei löydetä olemassaolevaa taskia, kutsu palauttaa aina arvon *RET_NOEXIST*. Jos taski taas löydetään, mutta signaalin numero ei ole sallittu (kaikki sallitut signaalit on määritelty yllämainituilla vakioilla), kutsu palauttaa aina arvon *RET_INVAL*.

Signaali *SIG_STAT* ei vaikuta millään tavalla taskin tilaan, mutta sen avulla voidaan tarkistaa, missä tilassa taski on tai onko se edes olemassa. Jos *signal*-kutsulla lähetetään *SIG_STAT*-signaali olemassaolevalle taskille, kutsu palauttaa taskin sen hetkisen tilan. Tila voi olla *TASK_RUNNING*, *TASK_STOPPED* tai *TASK_SLEEPING*. Jos taskia ei ole olemassa (eli sen tila on *TASK_KILLED*), kutsu palauttaa paluuarvon *RET_NOEXIST*.

Kuvassa 3.1 on esitelty taskin tilan vaihtuminen, kun sille lähetetään signaali tai kutsutaan järjestelmäkutsuja *create*, *sleep* ja *exit*. Myös käyttöjärjestelmän lähettämä ajastinsignaali on tyypiltään *SIG_CONT*. Kuvaan on merkitty myös muteksien ja semaforien käyttöön liittyvien järjestelmäkutsujen vaikutus taskin tilaan.

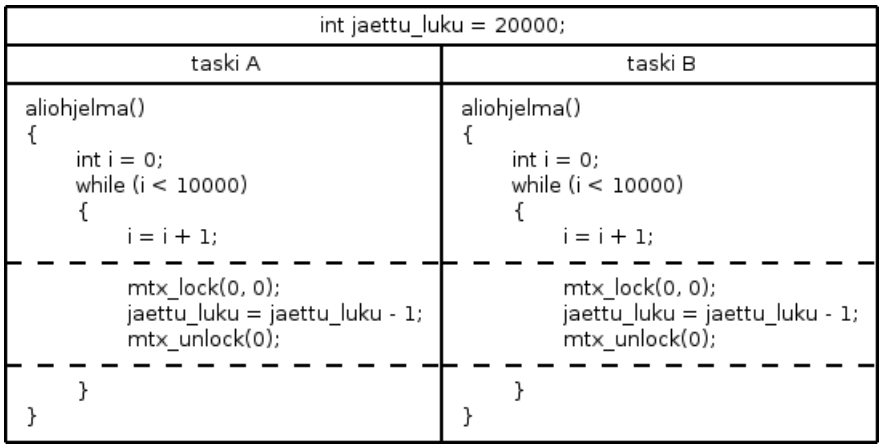


Kuva 3.1: Taskin tilakaavio

4 MUTEKSIT

Muteksi on yksinkertainen synkronointimekanismi, jonka avulla voidaan toteuttaa kriittisiä alueita, joita ei pidä suorittaa samanaikaisesti monessa taskissa. Kriittisten alueiden sisälle voidaan sijoittaa esimerkiksi sellaisten muistialueiden tai muiden resurssien käsittely, joita taskien on luettava tai kirjoitettava vuorotellen, jotta ohjelma toimisi odotetulla tavalla. Tapausta, jossa tiettyyn ohjelman osaan saa päästä vain yksi taski kerrallaan, kutsutaan poissulkevuusongelmaksi (mutual exclusion problem).

Eräs tyypillinen esimerkki poissulkevuusongelmasta on tilanne, jossa kaksi rinnakkain suoritettavaa taskia kasvattaa tai pienentää yhteisen muuttujan arvoa. Ohjelma on kuvassa 4.1 nähtävän C-kielisen ohjelman tyyppinen, taskit A ja B suorittavat molemmat samaa aliohjelmaa. Kriittinen alue on merkitty katkoviivoilla.



Kuva 4.1: Kriittinen alue

Molemmat taskit siis suorittavat samaa tai samanlaista ohjelmaa ja ne käsittelevät yhteistä *jaettu_ luku* -nimistä muuttujaa. Muuttuja alustetaan käynnistysvaiheessa arvoon 20000. Kumpikin taski suorittaa while-silmukkaa 10000 kertaa ja jokaisella iteraatiolla yhteisen muuttujan arvoa pienennetään yhdellä. Muuttujan arvon pitäisi siis olla nolla, kun molemmat taskit ovat päättyneet.

Oletetaan, että kuvassa näkyviä *mtx_ lock*- ja *mtx_ unlock*-kutsuja ei ole. C-kielistä lähdekoodia tarkastelemalla ohjelma näyttää toimivan täysin oikein ja se toimiikin, jos taskeja ei suoriteta rinnakkain, eli taski B käynnistetään vasta siinä vaiheessa kun taski A on jo päättynyt. Ohjelman assemblykielistä versiota tarkastelemalla voitaisiin kuitenkin nähdä, että muuttujan arvon pienentäminen ei tapahdu yhdellä suorittimen käskyllä. Muuttujan arvo on ensin luettava käyttömuistista johonkin työrekisteriin, jonka jälkeen työrekisterin arvoa pienennetään ja se tallennetaan takaisin käyttömuistiin.

Ongelma muodostuu, jos molemmat taskit suorittavat samanaikaisesti muuttujan arvoa pienentävää ohjelman osaa. Esimerkiksi taski A on voinut lukea muuttujan arvon työrekisteriin ja heti sen jälkeen käyttöjärjestelmä suorittaa taskinvaihdon. Taski B pääsee nyt ajovuoroon ja suorittaa silmukkaa, kunnes ajovuoroon siirretään taas taski A, joka edellisellä taskinvaihdolla keskeytettiin, kun muuttujan arvoa oltiin pienentämässä.

Nyt taski A:n suoritus jatkuu edellisestä paikasta, eli se on jo lukenut muuttujan arvon käyttömuistista työrekisteriin ja pienentää rekisterin arvoa nyt yhdellä sekä tallentaa rekisterin takaisin työmuistiin. Yhteisen muuttujan arvo on kuitenkin muuttunut, koska taski B oli välillä ajovuorossa. Taski A ei tiedä sitä vaan käyttää edelleen vanhaa jo luettua arvoa, pienentää sitä ja tallentaa takaisin muistiin. Taski B:n ajovuorolla suoritetut operaatiot siis mitätöityvät.

Ratkaisu ongelmaan on muuttujan pienennysoperaation sijoittaminen kriittisen alueen sisään, jolloin taskin suoritus keskeytyy sen yrittäessä päästä kriittiselle alueelle, jos toinen taski on jo sen sisällä. Keskeytetyn taskin suoritus voi jatkua vasta sitten, kun toinen taski poistuu kriittiseltä alueelta. Kriittinen alue voidaan luoda muteksin avulla. Kriittiselle alueelle tultaessa muteksi lukitaan järjestelmäkutsulla *mtx_lock* ja kriittiseltä alueelta poistuttaessa se on avattava kutsulla *mtx_unlock*.

Kutsulla *mtx_lock* on kaksi parametria: ensimmäisellä määritellään lukittavan muteksin numero ja toisella se, onko lukituksen oltava odottava vai ei-odottava. Jos muteksia ei ole vielä lukittu, se lukitaan ja taskin suoritus jatkuu. Jos muteksi on jo lukittu, taskin suoritus pysähtyy välittömästi ja jatkuu vasta, kun toinen muteksin jo aiemmin lukinnut taski avaa lukituksen. Muteksin lukitseva taski merkitään muteksin omistajaksi ja mikään muu taski ei voi avata sitä.

Aiemmin lukitun muteksin uusi lukitusyritys ei kuitenkaan johda taskin pysäyttämiseen, jos lukitus on määritelty ei-odottavaksi. Tällöin sen suoritus jatkuu, mutta kutsu palauttaa virhenumeron *RET_LOCKED*. Tällöin taski voi suorittaa välillä muita tehtäviä, mutta käyttäjän on itse ohitettava kriittinen alue, jos kutsun paluuarvo kertoo muteksin olevan lukittu.

On tärkeää huomata, että valmiiksi lukitun muteksin lukitsemisyritys johtaa taskin pysäyttämiseen myös silloin, kun muteksi on jo aiemmin lukittu saman taskin toimesta. Tällöin syntyy niin sanottu deadlock-tilanne, eli taski odottaa muteksin vapautumista ikuisesti, koska mikään muu taski ei voi avata taskin jo omistamaa muteksia. Tällainen tilanne voidaan purkaa vain lähettämällä jumiutuneelle taskille *SIG_CONT*-signaali tai tuhoamalla taski *SIG_KILL*-signaalilla.

Järjestelmäkutsu *mtx_unlock* avaa argumenttina annetun muteksin ja sitä käytetään pareittain järjestelmäkutsun *mtx_lock* kanssa. Kutsu avaa muteksin ja käynnistää muteksia odottavan (lukituksen aikana *mtx_lock*-kutsun suorittaneen) taskin. Jos muteksia odottavia taskeja on useita, niistä herätetään se, jonka vuorottaja ensimmäisenä löytää. Odottavia taskeja ei siis ole millään tavalla priorisoitu järjestelmän nykyisessä versiossa.

Edellämainittu kriittinen alue voidaan toteuttaa myös seuraavassa kappaleessa esiteltävillä semaforeilla. Muteksi ja binäärinen semafori muistuttavatkin toisiaan, mutta niillä on joitain olennaisia eroja. Mutekseista poiketen semaforeilla ei ole omistajaa, joten lukitun semaforin voi avata mikä tahansa taski. Poissulkevuusongelman lisäksi semaforeja voidaan siis käyttää myös taskien väliseen kommunikaatioon. Koska mutekseilla on omistaja, niillä voi olla muutamia lisäominaisuuksia, joita semaforeilla ei ole /3/.

Rekursiivinen muteksi voidaan lukita uudelleen omistavan taskin toimesta ja lukitus avataan vasta, kun avauksen suorittavaa järjestelmäkutsua on kutsuttu yhtä monta kertaa. Rekursiivinen muteksi sisältää siis laskurin, jonka arvo kasvaa aina lukittaessa ja pienenee avattaessa.

Rekursiivisen muteksin avulla voidaan toteuttaa aliohjelmia, jotka lukitsevat uudelleen saman muteksin, jonka niitä kutsuva pääohjelma on jo lukinnut. Kutsujan ei siis tarvitse välittää muteksin sen hetkisestä tilasta, koska aliohjelma ei lukkiudu, vaikka sama muteksi olisi jo lukittu. Toisaalta rekursiiviset muteksit saattavat aiheuttaa vaikeasti löydettäviä ohjelmointivirheitä ja ne myös pidentävät muteksin lukitusaikaa. Järjestelmän tämänhetkisessä versiossa ei ole rekursiivisia mutekseja.

Myöskään toista muteksien lisäominaisuutta ei ole tällä hetkellä toteutettu, mutta siihen on varauduttu erottamalla muteksit ja semaforit toisistaan. Ominaisuudelle ei ole vielä tarvetta, koska taskeilla ei ole prioriteetteja.

Ominaisuus liittyy priorisoitujen taskien synkronointiin liittyvään ongelmatilanteeseen, jossa matalalla prioriteetilla varustettu taski on lukinnut muteksin ja estää siten korkean prioriteetin taskin suorituksen. Ongelmaa kutsutaan prioriteetti-inversioksi (priority inversion) ja se voi syntyä, kun järjestelmä suorittaa vähintään kolmea taskia toisistaan poikkeavilla prioriteeteilla /4/.

Korkean prioriteetin taski odottaa matalan prioriteetin taskin lukitseman muteksin avaamista, mutta prioriteetiltaan näiden väliin sijoittuva kolmas taski keskeyttää hetkellisesti matalimman prioriteetin taskin suorituksen ja siten myös prioriteetiltaan korkeimman taskin suoritus viivästyy. Ongelma voidaan kiertää esimerkiksi korottamalla muteksin omistaman taskin prioriteettia väliaikaisesti, jotta se pääsee avaamaan korkean prioriteetin taskin odottaman muteksin.

5 SEMAFORIT

Semafori on kolmas järjestelmän tukema mekanismi taskien synkronointiin. Sen avulla voidaan ratkaista edellisessä kappaleessa mainittu poissulkevuusongelma, mutta mutekseista poiketen sillä on myös muita käyttötarkoituksia. Koska mikä tahansa taski voi avata lukitun semaforin, voidaan sen avulla ohjata taskin suoritusta toisesta taskista käsin kuten signaaleilla.

Semaforit voivat olla binäärisiä tai laskevia. Binäärisellä semaforilla on nimensä mukaisesti vain kaksi tilaa, lukittu ja ei-lukittu. Laskeva semafori lukittuu vasta kun semaforiin sisältyvän laskurin arvo muuttuu negatiiviseksi. Jos taski yrittää lukita semaforin, jonka arvo on jo negatiivinen, taski pysäytetään ja suoritus jatkuu vasta, kun toinen taski avaa lukituksen korottamalla semaforin arvoa. Arvoa muuttavista operaatioista käytetään usein nimiä *wait* ja *post*.

Semaforeja käytetään usein tuottaja-kuluttaja -ongelman ratkaisuun. Kyseisessa tapauksessa toinen taskeista toimii tietoa vastaanottavana kuluttajana ja toinen tietoa lähettävänä tuottajana. Tieto voidaan lähettää taskilta toiselle esim. 16 tavun pituisen puskurin välityksellä. Taskien suoritus ei välttämättä etene samalla nopeudella, joten ne on synkronoitava, jotta puskuri ei koskaan tyhjene kokonaan tai täyty liikaa (ylivuoto).

Kuluttaja odottaa semaforia järjestelmäkutsulla *sem_ wait* ja semaforin arvo pienee yhdellä aina, kun kutsu suoritetaan. Kun semaforin arvo muuttuu negatiiviseksi, se lukittuu ja taski pysäytetään. Näin varmistetaan, että puskurista luetaan tietoa vain niin monta tavua, kuin tuottaja on sinne tallentanut. Kutsulla on kaksi parametria: semaforin numero ja lukituksen tyyppi, joka voi *mtx_ lock*-kutsun mukaisesti olla odottava tai ei-odottava.

Tuottaja tallentaa puskuriin tietoa ja kutsuu järjestelmäkutsua *sem_post* ja antaa näin kuluttajataskille luvan edetä ja lukea tietoa puskurista. Edellisen kappaleen poissulkevuusongelmasta poiketen vain toinen taskeista odottaa (lukitsee) semaforin, ja toinen taski avaa sen. Kutsulla *sem_post* on kaksi parametria: semaforin numero ja kuluttajalle annettavien ”lupien” määrä, eli luku, jolla semaforin arvoa kasvatetaan. Jos toinen argumentti on 8, kuluttajataski voi suorittaa *sem_ wait*-kutsun kahdeksan kertaa, ennen kuin semafori lukkiutuu ja taski pysähtyy.

Tuottajana toimiva taski voi pyytää järjestelmältä semaforin nykyisen arvon järjestelmäkutsulla *sem_ get*. Näin tuottaja voi tarkistaa, onko puskurissa tilaa, eli onko kuluttaja ehtinyt jo lukemaan sinne aiemmin tallennetun tiedon. Jos tuottajalla ei ole puskurin täytyttyä muita tehtäviä, se voi luovuttaa ajovuoron *schedule*-kutsulla. Puskurina käytetään usein rengaspuskuri-tyyppistä tietorakennetta, myös linkitetty lista on mahdollinen.

Parempi ratkaisu edellä mainittuun ongelmaan on käyttää kahta semaforia, joista toinen ilmoittaa puskuriin tallennetun tiedon määrän ja toinen puskurin vapaan tilan. Tuottajana toimiva taski pienentää ensin vapaan tilan ilmoittavan semaforin arvoa ja pysähtyy, jos tilaa ei ole. Seuraavaksi tuottaja tallentaa tiedon puskuriin ja antaa kuluttajalle luvan edetä kasvattamalla tallennetun tiedon määrää ilmoittavaa semaforia.

Kuluttaja vastaavasti pienentää ensin tallennetun tiedon määrää ilmoittavaa semaforia, lukee tuottajan lähettämän tiedon ja kasvattaa vapaan tilan määrää ilmoittavaa semaforia. Kuva 5.1 esittää tämän tyyppistä ratkaisua, tietoa siirretään tavu kerrallaan ja sen siirtämiseen käytetään molemmille taskeille yhteistä *jaettu_ data* muuttujaa. Vapaata tilaa alustava semafori alustetaan arvoon 1 kasvattamalla sen arvoa *sem_ post* kutsulla. Järjestelmä alustaa semaforit oletuksena arvoon 0.

```
char jaettu_data = 0;
char vapaa = 0;
char tallennettu = 1;
sem_post(vapaa, 1);
```

tuottaja	kuluttaja

tuottaja()
{
  while (1)
  {
  sem_wait(vapaa, 0);
  jaettu_data = tuota_dataa();
  sem_post(tallennettu, 1)
  }
}
	kuluttaja()
	{
	  while (1)
	  {
	  sem_wait(tallennettu, 0);
	  kayta_dataa(jaettu_data);
	  sem_post(vapaa, 1);
	  }
	}

Kuva 5.1: Tuottaja-kuluttaja -ongelma

Semaforien ja muteksien avulla voidaan ratkaista monia muitakin rinnakkaisohjelmointiin liittyviä ongelma. Klassisia esimerkkejä ovat esimerkiksi Suhas S. Patilin vuonna 1971 esittelemä *cigarette smokers problem* /5/ ja semaforien keksijän, hollantilaisen Edsger Wybe Dijkstran semaforien käyttöä koskevasta artikkelista peräisin oleva *sleeping barber problem* /6, s. 80/

6 SUUNNITELTUJA LISÄOMINAISUUKSIA

Käyttöjärjestelmä toimii tällä hetkellä melko hyvin, mutta se on kuitenkin vasta varhainen kehitysversio. Järjestelmän käyttöönotto jossain oikeassa sovelluksessa edellyttää laajamittaista testausta, sillä tämänhetkiset testiohjelmat testaavat vain muutamia perusominaisuuksia, eikä niitäkään testata kattavasti.

Järjestelmään on jo suunniteltu lisäominaisuuksia, osa lisäominaisuuksista ei välttämättä ole kovin hyödyllisiä sulautettujen järjestelmien tyypillisissä käyttötapauksissa, mutta ne aiotaan kuitenkin toteuttaa opiskelutarkoituksessa.

Prioriteetit lisäämällä järjestelmästä tulisi tehokkaampi ja paremmin aikakriittisiin käyttökohteisiin soveltuva. Prioriteetit voitaisiin toteuttaa esimerkiksi lisäämällä ne nykyisen kiertovuorottelun rinnalle. Tällöin korkeamman prioriteetin taskit olisivat etusijalla, mutta saman prioriteetin jakavia taskeja käsiteltäisiin edelleen kiertovuorotteluna.

Taskit on tarkoitus järjestellä esim. Unix-järjestelmistä tuttuun puurakenteeseen. Tällöin jokaisella taskilla olisi vapaavalintainen määrä alataskeja (child) ja yksi ylätaski (parent). Puurakenteen toteuttaminen tehokkaasti edellyttäisi kiinteän kokoisia taulukkoja edistyneempiä tietorakenteita, kuten linkitettyjä listoja.

Tiedostot lisäämällä ulkoiset muistit, merkkipohjaiset näytöt, ym. laitteet voitaisiin abstrahoida yksinkertaisen rajapinnan avulla käsiteltäviksi. Esimerkiksi merkkijonon kirjoittaminen näytölle tapahtuisi avaamalla näyttöä vastaava tiedosto. Tiedostot mahdollistaisivat myös tiedostojen lukemisen ulkoisesta muistista kuten muistikortilta ja muistikortilla sijaitsevien valmiiksi käännettyjen ohjelmien suorittamisen. Ominaisuus mahdollistaisi myös järjestelmää hyödyntävän laitteen ohjelmiston päivittämisen tietokoneen sarjaportin välityksellä tai jopa langattomasti.

Järjestelmän käynnistämisessä voitaisiin hyödyntää AVR-mikro-ohjaimen käynnistyssegmenttiä (boot segment). Tällöin käyttöjärjestelmä käynnistyisi aina automaattisesti kun virrat kytketään, eikä sitä tarvitsisi käynnistää *init* kutsulla pääohjelmasta käsin. Ohjelmamuistiin voidaan kirjoittaa vain käynnistyssegmentissä sijaitsevasta ohjelmasta käsin, joten ominaisuus on muutenkin tarpeellinen.

7 YHTEENVETO

Koska käyttöjärjestelmä toteutettiin itsenäisesti käytännössä harrasteprojektina, ei järjestelmän ominaisuuksia ollut mitenkään etukäteen määritelty. Järjestelmään pyrittiin kuitenkin sisällyttämään suoritinteholtaan ja muistitilaltaan vaatimattomalla mikro-ohjaimella toimivan järjestelmän perusominaisuudet, ja tässä onnistuttiinkin melko hyvin.

Monet järjestelmään alunperin suunnitellut ominaisuudet, kuten tiedostot jäivät kuitenkin vielä toteuttamatta. Voidaankin sanoa, että jos järjestelmän kehitystä jatketaan, ovat kaikkein haastavimmat ja mielenkiintoisimmat asiat vielä jäljellä.

Järjestelmän toteuttaminen on vienyt melko paljon aikaa, koska suuri osa alkuperäisistä ratkaisuista on kehityksen edetessä hylätty ja koko järjestelmä onkin käytännössä kirjoitettu kahteen kertaan kokonaan uudestaan. Monia virheitä olisi voitu välttää, jos assembly-ohjelmointitaidot olisivat olleet paremmat. Esimerkiksi taskien tietorakenteet toteutettiin aluksi epäkäytännöllisellä tavalla, joka teki järjestelmästä tarpeettoman monimutkaisen ja myös hitaamman.

Työ on kokonaisuutena opettanut paljon sekä matalan tason ohjelmoinnista, että myös käyttöjärjestelmistä yleisesti. Moniin käyttöjärjestelmien konsepteihin tutustuttiin ensin Linux-ympäristössä, että niiden toimintaperiaatteet opittiin ymmärtämään. Työ on siis samalla tehnyt jo olemassaolevissa käyttöjärjestelmissä olevista prosessien synkronointimekanismeista ja rinnakkaisohjelmoinnista helppotajuisempia.

LÄHTEET

1 Introduction To Unix Signal Programming
[online][viitattu 2.12.2011]
http://users.evtek.fi/ tk/rtp/signals-programming.html

2 The GNU C Library - Signal Handling
[online][viitattu 2.12.2011]
http://www.cs.utah.edu/dept/old/texinfo/glibc-manual-0.02/library_21.html

3 Mutexes and Semaphores Demystified | Embedded Systems Experts
[online][viitattu 2.12.2011]
http://www.netrino.com/node/202

4 Introduction to Priority Inversion
[online][viitattu 2.12.2011]
http://www.netrino.com/node/74

5 Rinnakkainen ohjelmointi, Semaforit
[online][viitattu 2.12.2011]
http://www.tol.oulu.fi/users/ari.vesanen/Rinn_Ohjelm/Luennot/Semaforit.html

6 Andrew S. Tanenbaum 1997. Operating Systems - Design and Implementation, Second Edition

LIITTEET

Liite 1. Käyttöjärjestelmän otsikkotiedosto

Liite 2. Käyttöjärjestelmän otsikkotiedosto C-kielen prototyypeille

Liite 3. Käyttöjärjestelmän lähdekoodi

Liite 4. Testiohjelman lähdekoodi

Liite 5. Signaalitestin lähdekoodi

Liite 6. Semaforitestin lähdekoodi

Liite 7. Muteksitestin lähdekoodi

Liite 8. Makefile järjestelmän ja testiohjelmien kääntämiseen

```
#define CPU_FREQ 16000000       /* cpu clock frequency (Hz) */
#define SCHED_FREQ 1000         /* scheduler frequency (Hz) */

#define TASK_CNT 5              /* number of processes */
#define STACK_SIZE 200          /* size of stack in bytes */

#define SEM_CNT 5               /* number of semaphores */
#define MTX_CNT 5               /* number of mutexes */

#define TASK_KILLED 0           /* task is killed */
#define TASK_RUNNING 1          /* task is running */
#define TASK_STOPPED 2          /* task is stopped (by sending stop signal) */
#define TASK_SLEEPING 3         /* task is sleeping (waiting for signal, timer, semaphore or mutex) */

#define SIG_STAT 0              /* ask state of process, can be used for checking if task exists */
#define SIG_KILL 1              /* kill task */
#define SIG_STOP 2              /* stop task */
#define SIG_CONT 3              /* continue task */

#define RET_OK 0                /* no error */
#define RET_INVAL -128          /* invalid argument */
#define RET_LOCKED -127         /* semaphore or mutex is already locked */
#define RET_INTR -126           /* lock was interrupted by signal */
#define RET_NOEXIST -125        /* resource not found */
#define RET_NOAVAIL -124        /* resource not available */
#define RET_PERM -123           /* operation not permitted */
```

```
#include "avros_asm.h"

void init(void*);
void lock();
void unlock();
void schedule();
unsigned long get_time();
char create(void*);
void exit();
char get_id();
unsigned int sleep(unsigned int time);
char signal(char id, unsigned char sig);
char sem_wait(unsigned char sem, unsigned char trywait);
char sem_post(unsigned char sem, unsigned char cnt);
char sem_get(unsigned char sem);
char mtx_lock(unsigned char mtx, unsigned char trylock);
char mtx_unlock(unsigned char mtx);
```

```
#include <avr/io.h>
#include "avros_asm.h"
        ;; temp storage registers
#define tmp r18
#define tmp2 r19
#define tmp3 r20
#define tmp4 r21
        ;; system call argument registers
#define arg1 r24
#define arg2 r25
#define arg3 r22
#define arg4 r23
        ;; system call return value registers
#define ret1 r24
#define ret2 r25
#define ret3 r22
#define ret4 r23
        ;; data type sizes
#define BYTE 1
#define WORD 2
        ;; task structure byte offsets
#define TASK_ID 0
#define TASK_STATE (TASK_ID + BYTE)
#define TASK_LOCK (TASK_STATE + BYTE)
#define TASK_TIMER (TASK_LOCK + BYTE)
#define TASK_STACK_PTR (TASK_TIMER + WORD)
#define TASK_STACK (TASK_STACK_PTR + WORD)
#define TASK_SIZE (TASK_STACK + STACK_SIZE)
#define TASK_STACK_BOTTOM (TASK_SIZE - 1)
#define TASK_CONTEXT_SIZE 35
#define TASK_ADDR_IDLE (tasks + TASK_SIZE * TASK_CNT)
        ;; identifiers for lock types
#define LOCK_SEM 0b01000000
#define LOCK_MTX 0b10000000
        ;; value to add to timer counter when preparing next scheduler call
#define TCNT_ADD CPU_FREQ / (SCHED_FREQ * 256)

        .data                                   ; data segment begins
tasks:  .skip TASK_SIZE * (TASK_CNT + 1)        ; task array, one extra slot for idle task
task_addr: .skip WORD                           ; address of current task
atomic_intr: .skip BYTE                         ; atomic block interruption flag
sys_time: .skip BYTE * 4                        ; four bytes system time
semaphores: .skip BYTE * SEM_CNT                ; semaphore array
mutexes: .skip BYTE * MTX_CNT                   ; mutex array

        .text                                   ; code segment begins
        ;; store immediate byte to i/o at direct address
        .macro outi addr, val, tmp
        ldi \tmp, \val
        out \addr, \tmp
        .endm

        ;; store immediate byte to data space at indirect address
        .macro sti addr, val, tmp
        ldi \tmp, \val
        st \addr, \tmp
        .endm

        ;; store immediate byte to data space at direct address
        .macro stsi addr, val, tmp
        ldi \tmp, \val
        sts \addr, \tmp
        .endm

        ;; store immediate byte to data space at indirect address with displacement
        .macro stdi addr, val, tmp
        ldi \tmp, \val
        std \addr, \tmp
        .endm

        ;; load immediate word to register pair
        .macro ldiw regl, regh, val
        ldi \regl, lo8(\val)
        ldi \regh, hi8(\val)
```

```
        .endm

        ;; add immediate word to register pair
        .macro addiw regl, regh, val
        subi \regl, lo8(-(\val))
        sbci \regh, hi8(-(\val))
        .endm

        ;; enable output compare interrupt for timer 0
        .macro oci_enable
        in tmp, _SFR_IO_ADDR(TIMSK)             ; load current flags
        sbr tmp, 1 << OCIE0                     ; add OCIE0 flag
        out _SFR_IO_ADDR(TIMSK), tmp            ; store new flags
        .endm

        ;; disable output compare interrupt for timer 0
        .macro oci_disable
        in tmp, _SFR_IO_ADDR(TIMSK)             ; load current flags
        cbr tmp, 1 << OCIE0                     ; remove OCIE0 flag
        out _SFR_IO_ADDR(TIMSK), tmp            ; store new flags
        .endm

        ;; begin atomic block, store global interrupt flag
        .macro ATOMIC_BEGIN
        brid 1001f                              ; step over if interrupts disabled
        stsi atomic_intr, 1, tmp
        rjmp 1002f
1001:
        stsi atomic_intr, 0, tmp
1002:
        cli
        .endm

        ;; end atomic block, restore global interrupt flag to previous state, return
        .macro ATOMIC_END
        lds tmp, atomic_intr
        cpi tmp, 0
        breq 1101f
        reti                                    ; enable interrupts and return
1101:
        ret                                     ; return
        .endm

        ;; initialize array of size at address to value
        .macro memset addr, val, size, cnt, tmp
        ldi \cnt, 0                             ; set counter
        ldiw ZL, ZH, \addr                      ; store array address to Z
        rjmp 1202f                              ; jump to counter comparison
1201:
        sti Z+, \val, \tmp                      ; store value to array
        inc \cnt                                ; increase counter
1202:
        cpi \cnt, \size                         ; compare counter
        brlo 1201b                              ; next iteration
        .endm

        ;; save task execution context
        .macro save
        push r0                                 ; push r0
        in r0, _SFR_IO_ADDR(SREG)               ; load status register
        cli                                     ; disable interrupts
        push r0                                 ; push status register
        push r1
        push r2
        push r3
        push r4
        push r5
        push r6
        push r7
        push r8
        push r9
        push r10
        push r11
        push r12
```

```
        push r13
        push r14
        push r15
        push r16
        push r17
        push r18
        push r19
        push r20
        push r21
        push r22
        push r23
        push r24
        push r25
        push r26
        push r27
        push r28
        push r29
        push r30
        push r31
        .endm

        ;; restore task execution context
        .macro restore
        pop r31
        pop r30
        pop r29
        pop r28
        pop r27
        pop r26
        pop r25
        pop r24
        pop r23
        pop r22
        pop r21
        pop r20
        pop r19
        pop r18
        pop r17
        pop r16
        pop r15
        pop r14
        pop r13
        pop r12
        pop r11
        pop r10
        pop r9
        pop r8
        pop r7
        pop r6
        pop r5
        pop r4
        pop r3
        pop r2
        pop r1
        pop r0                                  ; pop status register
        out _SFR_IO_ADDR(SREG), r0              ; store status register
        pop r0                                  ; pop r0
        .endm


        ;; initialize operating system
        .global init
init:
        ;; initialize tasks, all tasks killed by default
        clr tmp                                 ; init task counter
        ldiw ZL, ZH, tasks                      ; load address of task array to Z
        rjmp init_2                             ; counter comparison
init_1:
        inc tmp                                 ; increase task counter
        std Z + TASK_ID, tmp                    ; store id
        stdi Z + TASK_STATE, TASK_KILLED, tmp2  ; store state
        stdi Z + TASK_LOCK, 0, tmp2             ; store lock number
        stdi Z + TASK_TIMER, 0, tmp2            ; store timer lo byte
        stdi Z + TASK_TIMER + 1, 0, tmp2        ; store timer hi byte
```

```
        stdi Z + TASK_STACK_PTR, 0, tmp2        ; store stack pointer lo byte
        stdi Z + TASK_STACK_PTR + 1, 0, tmp2    ; store stack pointer hi byte
        addiw ZL, ZH, TASK_SIZE                 ; add offset to next task
init_2:
        cpi tmp, TASK_CNT + 1                   ; compare task counter
        brne init_1                             ; next iteration
        ;; initialize semaphores
        memset semaphores, 1, (BYTE * SEM_CNT), tmp, tmp2
        ;; initialize mutexes
        memset mutexes, 0, (BYTE * MTX_CNT), tmp, tmp2
        stsi atomic_intr, 0, tmp                ; clear atomic block interruption flag
        memset sys_time, 0, (BYTE * 4), tmp, tmp2 ; clear system time
        ;; create idle task
        rcall create_idle
        ;; store address of last task to task pointer,
        ;; first task slot will be used when creating main task
        stsi task_addr, lo8(tasks + TASK_SIZE * (TASK_CNT - 1)), tmp
        stsi task_addr + 1, hi8(tasks + TASK_SIZE * (TASK_CNT - 1)), tmp
        rcall create                            ; create main task
        ldd tmp, Z + TASK_STACK_PTR             ; load lower byte of task stack pointer
        ldd tmp2, Z + TASK_STACK_PTR + 1        ; load upper byte of task stack pointer
        ;; fix stack pointer, main task need no preallocated context space
        addiw tmp, tmp2, (TASK_CONTEXT_SIZE - 2)
        out _SFR_IO_ADDR(SPL), tmp              ; store lower byte of cpu stack pointer
        out _SFR_IO_ADDR(SPH), tmp2             ; store upper byte of cpu stack pointer
        sts task_addr, ZL                       ; store address of main task to task pointer
        sts task_addr + 1, ZH                   ;
        outi _SFR_IO_ADDR(TCCR0), (1 << CS02), tmp ; set timer 0 speed to xtal / 256
        oci_enable                              ; enable output compare interrupt
        outi _SFR_IO_ADDR(TCNT0), 0, tmp        ; reset timer 0 counter
        outi _SFR_IO_ADDR(OCR0), TCNT_ADD, tmp  ; prepare first scheduler interrupt
        rcall switch                            ; switch to main task
        rjmp schedule                           ; schedule main task


        ;; prevent task scheduling
        .global lock
lock:
        ATOMIC_BEGIN                            ; begin atomic block
        oci_disable                             ; disable output compare interrupt
        ATOMIC_END                              ; end atomic block


        ;; allow task scheduling
        .global unlock
unlock:
        ATOMIC_BEGIN                            ; begin atomic block
        oci_enable                              ; enable output compare interrupt
        ATOMIC_END                              ; end atomic block


        ;; find task
find:
        ldi tmp3, TASK_CNT                      ; init task counter
        lds ZL, task_addr                       ; load address of current task to Z
        lds ZH, task_addr + 1                   ;
        rjmp find_3                             ; counter comparison
find_1:
        dec tmp3                                ; decrease task counter
        addiw ZL, ZH, TASK_SIZE                 ; move pointer to next task
        ldi tmp4, hi8(TASK_ADDR_IDLE)           ; bound check task pointer
        cpi ZL, lo8(TASK_ADDR_IDLE)             ;
        cpc ZH, tmp4                            ;
        brlo find_2                             ; step over if in bounds
        ldiw ZL, ZH, tasks                      ; reset task pointer
find_2:
        mov XL, ZL                              ; copy lower byte of task address to XL
        mov XH, ZH                              ; copy upper byte of task address to XH
        add XL, tmp2                            ; add offset to task struct
        clr tmp4                                ; clear tmp4 for adding zero with carry
        adc XH, tmp4                            ; add zero with carry
        ld tmp4, X                              ; load task state
        cp tmp, tmp4                            ; check if matching task
        brne find_3                             ; step over
```

```
        ret                                     ; ready task found, return
find_3:
        cpi tmp3, 0                             ; compare task counter
        brne find_1                             ; next iteration
        ldi ZL, lo8(TASK_ADDR_IDLE)             ; task not found
        ldi ZH, hi8(TASK_ADDR_IDLE)             ; set Z to address of idle task
        ret


        ;; perform a task switch from current to next task.
switch:
        pop tmp2                                ; store return address
        pop tmp3                                ;
        ;; store current cpu stack pointer to stack pointer of current task
        mov XL, ZL                              ; save Z, contains address of next task
        mov XH, ZH                              ;
        lds ZL, task_addr                       ; load address of current task to Z
        lds ZH, task_addr + 1                   ;
        in tmp, _SFR_IO_ADDR(SPL)               ; load lower byte of cpu stack pointer
        std Z + TASK_STACK_PTR, tmp             ; store lower byte of task stack pointer
        in tmp, _SFR_IO_ADDR(SPH)               ; load upper byte of cpu stack pointer
        std Z + TASK_STACK_PTR + 1, tmp         ; store upper byte of task stack pointer
        mov ZL, XL                              ; restore Z
        mov ZH, XH                              ;
        ;; set new cpu stack pointer
        ldd tmp, Z + TASK_STACK_PTR             ; load lower byte of task stack pointer
        out _SFR_IO_ADDR(SPL), tmp              ; store lower byte of cpu stack pointer
        ldd tmp, Z + TASK_STACK_PTR + 1         ; load upper byte of task stack pointer
        out _SFR_IO_ADDR(SPH), tmp              ; store upper byte of cpu stack pointer
        sts task_addr, ZL                       ; store address of current task
        sts task_addr + 1, ZH                   ;
        push tmp3                               ; restore return address
        push tmp2                               ;
        ret                                     ; return


        ;; process timers
timers:
        clr tmp                                 ; init task counter
        ldiw ZL, ZH, tasks                      ; load address of task array to Z
        rjmp timers_3                           ; counter comparison
timers_1:
        inc tmp                                 ; increase task counter
        ldd XL, Z + TASK_TIMER                  ; load lower byte of timer
        ldd XH, Z + TASK_TIMER + 1              ; load upper byte of timer
        ldi tmp2, 0                             ; check if timer value is zero
        cpi XL, 0                               ; compare lower byte
        cpc XH, tmp2                            ; compare upper byte
        breq timers_2                           ; step over
        sbiw XL, 1                              ; decrease timer value
        brne timers_2                           ; check if timer value decremented to zero
        stdi Z + TASK_STATE, TASK_RUNNING, tmp2 ; wake up task
timers_2:
        std Z + TASK_TIMER, XL                  ; store lower byte of timer
        std Z + TASK_TIMER + 1, XH              ; store upper byte of timer
        addiw ZL, ZH, TASK_SIZE                 ; add offset to next task
timers_3:
        cpi tmp, TASK_CNT                       ; compare task counter
        brne timers_1                           ; next iteration
        ret


        ;; increases system time
inc_time:
        ldi tmp2, 1                             ; load 1 to tmp
        lds tmp, sys_time                       ; load first byte
        add tmp, tmp2                           ; increment first byte
        sts sys_time, tmp                       ; store first byte
        clr tmp2                                ; clear tmp
        lds tmp, sys_time + 1                   ; load second byte
        adc tmp, tmp2                           ; increment second byte
        sts sys_time + 1, tmp                   ; store second byte
        lds tmp, sys_time + 2                   ; load third byte
        adc tmp, tmp2                           ; increment third byte
```

```
        sts sys_time + 2, tmp                   ; store third byte
        lds tmp, sys_time + 3                   ; load fourth byte
        adc tmp, tmp2                           ; increment fourth byte
        sts sys_time + 3, tmp                   ; store fourth byte
        ret


        ;; idle task, running only when there are no other tasks ready to run
idle_task:
        rjmp idle_task


        ;; schedule next task and process system clock and task timers,
        ;; driven by hardware interruption
        .global TIMER0_COMP_vect
TIMER0_COMP_vect:
        save                                    ; save task execution context
        rcall timers                            ; process timers
        rcall inc_time                          ; increase system time
        ldi tmp, TASK_RUNNING                   ; load task state to tmp
        ldi tmp2, TASK_STATE                    ; load task struct offset to tmp2
        rcall find                              ; find next task to run
        rcall switch                            ; switch to next task
        oci_enable                              ; enable output compare interrupt
        in tmp, _SFR_IO_ADDR(TCNT0)             ; load current timer value
        subi tmp, (-TCNT_ADD)                   ; add scheduler interval
        out _SFR_IO_ADDR(OCR0), tmp             ; store compare register value
        restore                                 ; restore task execution context
        reti                                    ; enable interrupts and execute task


        ;; schedule next task explicitly
        .global schedule
schedule:
        save                                    ; save task execution context
        ldi tmp, TASK_RUNNING                   ; load task state to tmp
        ldi tmp2, TASK_STATE                    ; load task struct offset to tmp2
        rcall find                              ; find next task to run
        rcall switch                            ; switch to next task
        restore                                 ; restore task execution context
        reti                                    ; enable interrupts and execute task


        ;; get value of scheduler tick counter
        .global get_time
get_time:
        lds ret3, sys_time                      ; load first byte
        lds ret4, sys_time + 1                  ; load second byte
        lds ret1, sys_time + 2                  ; load third byte
        lds ret2, sys_time + 3                  ; load fourth byte
        reti


        ;; create new task
        .global create
create:
        ATOMIC_BEGIN                            ; begin atomic block
        ldi tmp, TASK_KILLED                    ; load task state to tmp
        ldi tmp2, TASK_STATE                    ; load task struct offset to tmp2
        rcall find                              ; find free task slot
        ldi tmp2, hi8(TASK_ADDR_IDLE)           ; check task address
        cpi ZL, lo8(TASK_ADDR_IDLE)             ; compare lower byte of task address
        cpc ZH, tmp2                            ; compare upper byte of task address
        breq create_1                           ; step over
        mov XL, ZL                              ; copy address of task to X
        mov XH, ZH                              ;
        ;; add offset to top of stack of task, preserve space for execution context and return address
        addiw XL, XH, TASK_STACK_BOTTOM - TASK_CONTEXT_SIZE
        std Z + TASK_STACK_PTR, XL              ; store address to stack pointer of task
        std Z + TASK_STACK_PTR + 1, XH          ;
        addiw ZL, ZH, (TASK_STACK_BOTTOM - 1)   ; move Z to upper byte of return address
        st Z, arg2                              ; store upper byte of return address
        std Z + 1, arg1                         ; store lower byte of return address
        addiw ZL, ZH, -(TASK_STACK_BOTTOM - 1)  ; move Z back to task address
```

```
        stdi Z + TASK_STATE, TASK_RUNNING, tmp  ; wake up task
        ldd ret1, Z + TASK_ID                   ; load id of task to ret1
        clr ret2                                ; clear ret2
        ATOMIC_END                              ; end atomic block
create_1:
        ldiw ret1, ret2, RET_NOAVAIL            ; load RET_NOAVAIL to ret
        ATOMIC_END                              ; end atomic block


        ;; create idle task
create_idle:
        ldiw ZL, ZH, TASK_ADDR_IDLE             ; load address of idle task to Z
        ldiw XL, XH, TASK_ADDR_IDLE             ; load address of idle task to X
        ;; add offset to top of stack of task, preserve space for execution context and return address
        addiw XL, XH, TASK_STACK_BOTTOM - TASK_CONTEXT_SIZE
        std Z + TASK_STACK_PTR, XL              ; store address to stack pointer of task
        std Z + TASK_STACK_PTR + 1, XH          ;
        addiw ZL, ZH, (TASK_STACK_BOTTOM - 1)   ; move Z to upper byte of return address
        sti Z, hi8(gs(idle_task)), tmp          ; store upper byte of idle task address
        stdi Z + 1, lo8(gs(idle_task)), tmp     ; store lower byte of idle task address
        addiw ZL, ZH, -(TASK_STACK_BOTTOM - 1)  ; move Z back to task address
        stdi Z + TASK_STATE, TASK_RUNNING, tmp  ; wake up task
        ret


        ;; end task
        .global exit
exit:
        cli                                     ; begin atomic block
        lds ZL, task_addr                       ; load address of current task to Z
        lds ZH, task_addr + 1                   ;
        stdi Z + TASK_STATE, TASK_KILLED, tmp   ; kill task
        stdi Z + TASK_LOCK, 0, tmp              ; clear lock number
        rjmp schedule                           ; schedule next task


        ;; get id of current task
        .global get_id
get_id:
        ATOMIC_BEGIN                            ; begin atomic block
        lds ZL, task_addr                       ; load address of current task to Z
        lds ZH, task_addr + 1                   ;
        ldd ret1, Z + TASK_ID                   ; load id of task to ret1
        clr ret2                                ; clear ret2
        ATOMIC_END                              ; end atomic block


        ;; wait for signal from other task or scheduler
        .global sleep
sleep:
        ATOMIC_BEGIN                            ; begin atomic block
        lds ZL, task_addr                       ; load address of current task to Z
        lds ZH, task_addr + 1                   ;
        std Z + TASK_TIMER, arg1                ; store timer value
        std Z + TASK_TIMER + 1, arg2            ;
        stdi Z + TASK_STATE, TASK_SLEEPING, tmp ; suspend task
        rcall schedule                          ; schedule next task
        ldd XL, Z + TASK_TIMER                  ; load timer value
        ldd XH, Z + TASK_TIMER + 1              ;
        ldi tmp2, 0                             ; check if timer value is zero
        cpi XL, 0                               ; compare lower byte
        cpc XH, tmp2                            ; compare upper byte
        breq sleep_2                            ; step over if zero
        mov ret1, XL                            ; copy remaining value of timer to ret
        mov ret2, XH                            ;
        ATOMIC_END                              ; end atomic block
sleep_2:
        ldiw ret1, ret2, 666
        ATOMIC_END                              ; end atomic block


        ;; send signal to task with matching id
.global signal
signal:
```

```
        ATOMIC_BEGIN                            ; begin atomic block
        mov tmp, arg1                           ; copy task id to tmp
        ldi tmp2, TASK_ID                       ; copy task struct offset to tmp2
        rcall find                              ; find task with matching id
        ldi tmp2, hi8(TASK_ADDR_IDLE)           ; check task address
        cpi ZL, lo8(TASK_ADDR_IDLE)             ; compare lower byte of task address
        cpc ZH, tmp2                            ; compare upper byte of task address
        breq signal_6                           ; step over
        ldd tmp, Z + TASK_STATE                 ; load task state
        cpi tmp, TASK_KILLED                    ; check if task is alive
        breq signal_6                           ;
        cpi arg3, SIG_KILL                      ; check if SIG_KILL
        breq signal_1                           ;
        cpi arg3, SIG_STOP                      ; check if SIG_STOP
        breq signal_2                           ;
        cpi arg3, SIG_CONT                      ; check if SIG_CONT
        breq signal_3                           ;
        cpi arg3, SIG_STAT                      ; check if SIG_STAT
        breq signal_4                           ;
        rjmp signal_5                           ; no valid signal, invalid argument
signal_1:
        stdi Z + TASK_STATE, TASK_KILLED, tmp   ; kill task
        ldiw ret1, ret2, RET_OK                 ; load RET_OK to ret
        ATOMIC_END                              ; end atomic block
signal_2:
        stdi Z + TASK_STATE, TASK_STOPPED, tmp  ; suspend task
        ldiw ret1, ret2, RET_OK                 ; load RET_OK to ret
        ATOMIC_END                              ; end atomic block
signal_3:
        stdi Z + TASK_STATE, TASK_RUNNING, tmp  ; wake up task
        ldiw ret1, ret2, RET_OK                 ; load RET_OK to ret
        ATOMIC_END                              ; end atomic block
signal_4:
        ldd ret1, Z + TASK_STATE                ; load task state to ret1
        clr ret2                                ; clear ret2
        ATOMIC_END                              ; end atomic block
signal_5:
        ldiw ret1, ret2, RET_INVAL              ; load RET_INVAL to ret
        ATOMIC_END                              ; end atomic block
signal_6:
        ldiw ret1, ret2, RET_NOEXIST            ; load RET_NOEXIST to ret
        ATOMIC_END                              ; end atomic block


        ;; wait for semaphore
        .global sem_wait
sem_wait:
        ATOMIC_BEGIN                            ; begin atomic block
        cpi arg1, SEM_CNT                       ; bound check semaphore number
        brsh sem_wait_3                         ; step over if out of bound
        mov ZL, arg1                            ; load semaphore number to Z
        clr ZH                                  ;
        addiw ZL, ZH, semaphores                ; add address of semaphore array
        ld tmp, Z                               ; load value of semaphore
        cpi arg3, 0                             ; check if blocking wait (argument is zero)
        breq sem_wait_1                         ; step over if blocking wait
        cpi tmp, 0                              ; check if semaphore already locked (negative value)
        brmi sem_wait_5                         ; step over if already locked
sem_wait_1:
        dec tmp                                 ; decrease value of semaphore
        st Z, tmp                               ; store new value of semaphore
        brpl sem_wait_2                         ; step over if semaphore is not locked (positive value)
        lds ZL, task_addr                       ; load address of current task to Z
        lds ZH, task_addr + 1                   ;
        stdi Z + TASK_STATE, TASK_SLEEPING, tmp2 ; suspend task
        mov tmp2, arg1                          ; copy semaphore number to tmp
        subi tmp2, (-LOCK_SEM)                  ; add semaphore lock offset
        std Z + TASK_LOCK, tmp2                 ; store lock number
        rcall schedule                          ; schedule next task
        ldd tmp2, Z + TASK_LOCK                 ; load lock number
        cpi tmp2, 0                             ; check if interrupted by signal
        brne sem_wait_4                         ; step over if interrupted
sem_wait_2:
        ldiw ret1, ret2, RET_OK                 ; load RET_OK to ret
```

```
        ATOMIC_END                              ; end atomic block
sem_wait_3:
        ldiw ret1, ret2, RET_INVAL              ; load RET_INVAL to ret
        ATOMIC_END                              ; end atomic block
sem_wait_4:
        ldiw ret1, ret2, RET_INTR               ; load RET_INTR to ret
        ATOMIC_END                              ; end atomic block
sem_wait_5:
        ldiw ret1, ret2, RET_LOCKED             ; load RET_LOCKED to ret
        ATOMIC_END                              ; end atomic block


        ;; post to semaphore
        .global sem_post
sem_post:
        ATOMIC_BEGIN                            ; begin atomic block
        cpi arg1, SEM_CNT                       ; bound check semaphore number
        brsh sem_post_3                         ; step over
        cpi arg3, 1                             ; check increment count, must be at least one
        brlo sem_post_3                         ; step over
        mov ZL, arg1                            ; store semaphore number to Z
        clr ZH                                  ;
        addiw ZL, ZH, semaphores                ; add address of semaphore array
        ld tmp, Z                               ; load value of semaphore
        add tmp, arg3                           ; increase value of semaphore
        st Z, tmp                               ; store new value of semaphore
        mov tmp, arg1                           ; copy argument to tmp
        subi tmp, (-LOCK_SEM)                   ; add semaphore lock offset
        ldi tmp2, TASK_LOCK                     ; load task struct offset to tmp2
        rcall find                              ; find task waiting for semaphore
        ldi tmp2, hi8(TASK_ADDR_IDLE)           ; check task address
        cpi ZL, lo8(TASK_ADDR_IDLE)             ; compare lower byte of task address
        cpc ZH, tmp2                            ; compare upper byte of task address
        breq sem_post_2                         ; step over
        stdi Z + TASK_STATE, TASK_RUNNING, tmp  ; wake up task
        stdi Z + TASK_LOCK, 0, tmp              ; clear lock number
sem_post_2:
        ldiw ret1, ret2, RET_OK                 ; load RET_OK to ret
        ATOMIC_END                              ; end atomic block
sem_post_3:
        ldiw ret1, ret2, RET_INVAL              ; load RET_INVAL to ret
        ATOMIC_END                              ; end atomic block


        ;; get semaphore value
        .global sem_get
sem_get:
        ATOMIC_BEGIN                            ; begin atomic block
        cpi arg1, SEM_CNT                       ; bound check semaphore number
        brsh sem_get_1                          ; step over
        mov ZL, arg1                            ; copy semaphore number to Z
        clr ZH                                  ;
        addiw ZL, ZH, semaphores                ; add address of semaphore array
        ld ret1, Z                              ; load value of semaphore to ret1
        clr ret2                                ; clear ret2
        ATOMIC_END                              ; end atomic block
sem_get_1:
        ldiw ret1, ret2, RET_INVAL              ; load RET_INVAL to ret
        ATOMIC_END                              ; end atomic block

        ;; lock mutex
mtx_lock:
        ATOMIC_BEGIN                            ; begin atomic block
        cpi arg1, MTX_CNT                       ; bound check mutex number
        brsh mtx_lock_3                         ; step over if out of bound
        lds ZL, task_addr                       ; load address of current task to Z
        lds ZH, task_addr + 1                   ;
        ldd tmp2, Z + TASK_ID                   ; load task id
        mov ZL, arg1                            ; copy mutex number to Z
        clr ZH                                  ;
        addiw ZL, ZH, mutexes                   ; add address of mutex array
        ld tmp, Z                               ; load value of mutex
        cpi arg3, 0                             ; check if blocking wait (argument is zero)
        breq mtx_lock_1                         ; step over if blocking wait
```

```
        cpi tmp, 0                              ; check if mutex is already locked
        brne mtx_lock_5                         ; step over if already locked
mtx_lock_1:
        cpi tmp, 0                              ; check if mutex is locked
        breq mtx_lock_2                         ; step over if not locked
        lds ZL, task_addr                       ; load address of current task to Z
        lds ZH, task_addr + 1                   ;
        stdi Z + TASK_STATE, TASK_SLEEPING, tmp2 ; suspend task
        mov tmp2, arg1                          ; copy mutex number to tmp
        subi tmp2, (-LOCK_MTX)                  ; add mutex lock offset
        std Z + TASK_LOCK, tmp2                 ; store lock number
        rcall schedule                          ; schedule next task
        ldd tmp2, Z + TASK_LOCK                 ; load lock number
        cpi tmp2, 0                             ; check if interrupted by signal
        brne mtx_lock_4                         ; step over if interrupted
        ldiw ret1, ret2, RET_OK                 ; load RET_OK to ret
        ATOMIC_END                              ; end atomic block
mtx_lock_2:
        st Z, tmp2                              ; lock mutex, current task is owner of lock
        ldiw ret1, ret2, RET_OK                 ; load RET_OK to ret
        ATOMIC_END                              ; end atomic block
mtx_lock_3:
        ldiw ret1, ret2, RET_INVAL              ; load RET_INVAL to ret
        ATOMIC_END                              ; end atomic block
mtx_lock_4:
        stdi Z + TASK_LOCK, 0, tmp              ; clear lock index
        ldiw ret1, ret2, RET_INTR               ; load RET_INTR to ret
        ATOMIC_END                              ; end atomic block
mtx_lock_5:
        ldiw ret1, ret2, RET_LOCKED             ; load RET_LOCKED to ret
        ATOMIC_END                              ; end atomic block
        .global mtx_lock

        ;; unlock mutex
        .global mtx_unlock
mtx_unlock:
        ATOMIC_BEGIN                            ; begin atomic block
        cpi arg1, MTX_CNT                       ; bound check mutex number
        brsh mtx_unlock_3                       ; step over
        lds ZL, task_addr                       ; load address of current task to Z
        lds ZH, task_addr + 1                   ;
        ldd tmp2, Z + TASK_ID                   ; load task id
        mov ZL, arg1                            ; copy mutex number to Z
        clr ZH                                  ;
        addiw ZL, ZH, mutexes                   ; add address of mutex array
        ld tmp, Z                               ; load value of mutex
        cpi tmp, 0                              ; check if mutex is already unlocked
        breq mtx_unlock_2                       ; step over if unlocked
        cp tmp, tmp2                            ; check if mutex if locked by this task
        ;; step over if not locked by this task (locked by other task, operation not permitted)
        brne mtx_unlock_4
        sti Z, 0, tmp                           ; unlock mutex, not owned by any task (zero value)
        mov tmp, arg1                           ; copy argument to tmp
        subi tmp, (-LOCK_MTX)                   ; add mutex lock offset
        ldi tmp2, TASK_LOCK                     ; load task struct offset to tmp2
        rcall find                              ; find task waiting for mutex
        ldi tmp2, hi8(TASK_ADDR_IDLE)           ; check task address
        cpi ZL, lo8(TASK_ADDR_IDLE)             ; compare lower byte of task address
        cpc ZH, tmp2                            ; compare upper byte of task address
        breq mtx_unlock_2                       ; step over
        stdi Z + TASK_STATE, TASK_RUNNING, tmp  ; wake up task
        stdi Z + TASK_LOCK, 0, tmp              ; clear lock index
mtx_unlock_2:
        ldiw ret1, ret2, RET_OK                 ; load RET_OK to ret
        ATOMIC_END                              ; end atomic block
mtx_unlock_3:
        ldiw ret1, ret2, RET_INVAL              ; load RET_INVAL to ret
        ATOMIC_END                              ; end atomic block
mtx_unlock_4:
        ldiw ret1, ret2, RET_PERM               ; load RET_PERM to ret
        ATOMIC_END                              ; end atomic block

        ;; required for initialization of data segment
        .global __do_copy_data
```

```
#include <avr/io.h>
#include "avros.h"

void signal_test();
void mutex_test();
void semaphore_test();

#define SIGNAL_TEST 1
#define MUTEX_TEST 2
#define SEMAPHORE_TEST 3

int main()
{
        /* port a and b are outputs */
        DDRA = 0xff;
        DDRB = 0xff;
        int testnum = SIGNAL_TEST;
        switch (testnum)
        {
        case SIGNAL_TEST:
                /* init operating system with signal_test entry point */
                init(signal_test);
                break;
        case MUTEX_TEST:
                /* init operating system with mutex_test entry point */
                init(mutex_test);
                break;
        case SEMAPHORE_TEST:
                /* init operating system with semaphore_test entry point */
                init(semaphore_test);
                break;
        default:
                break;
        }
        return 0;
}
```

```
/*
This program creates fours tasks running same code and synchronizes them using signals. Only one
of tasks is running at each time. Task rises one bit indicating it's id in port a and then sleeps
for 100 scheduler ticks. When task has slept for 100 ticks, it wakes up next task and suspend itself
until previous task wakes it with signal. So task execution order is 0, 1, 2, 3, 0, 1, 2, 3, etc.
Main task creates four tasks, stores their identifiers to array and then wakes up first of them.
Then it sleeps for 5000 scheduler ticks, kills all four tasks and rises four upper bits in port a
indicating end of program.
*/

#include <avr/io.h>
#include "avros.h"

char tasks[4];                  /* identifiers of test tasks */
char main_task = 0;             /* identifier of main task */

#define TASK_COUNT 4

void signal_task()
{
        static char task_cnt = 0; /* static task count variable, shared between all tasks */
        char task_num = task_cnt; /* local task number (index) variable */
        task_cnt ++;              /* increase shared task count variable */
        signal(main_task, SIG_CONT); /* signal main task that task has initialized */
        sleep(0);                    /* sleep until waken by other task */
        while (1)
        {
                PORTA = 1 << task_num; /* indicate task number by rising corresponding bit in port a */
                sleep(100);            /* sleep for 1000 scheduler ticks */
                /* calculate number of next task */
                int task_next = task_num + 1;
                if (task_next == TASK_COUNT)
                {
                        task_next = 0;
                }
                /* wake up next task and sleep until waken by other task */
                signal(tasks[task_next], SIG_CONT);
                sleep(0);
        }
}

void signal_test()
{
        /* port a is output */
        DDRA = 0xff;
        main_task = get_id();   /* get id of main task */
        int i = 0;
        for (i = 0; i < TASK_COUNT; i ++)
        {
                tasks[i] = create(signal_task); /* create task */
                sleep(0);                       /* wait until task has initialized */
        }
        /* wake up first task */
        signal(tasks[0], SIG_CONT);
        /* sleep for 10000 scheduler ticks */
        sleep(5000);
        /* kill all tasks */
        for (i = 0; i < TASK_COUNT; i ++)
        {
                signal(tasks[i], SIG_KILL);
        }
        /* rise four upper bits in port a */
        PORTA |= 0xf0;
        /* sit in loop */
        while(1) {}
}
```

```
/*
This program creates two tasks which are both running their own instance of subroutine mutex_test_var.
Global variable mutex_test_var is initialized to value 20015. Both instances of subroutine decrement
value of mutex_test_var for 10000 times. When both tasks are finished, value of variable should be 15.
Main routine mutex_test waits until both tasks are finished and then outputs the variable to port a.
Decrementing is protected by critical section implemented with mutex. Only one tasks can enter
the critical section at one time. When other task tries to enter the critical section by calling
mtx_lock, it will be suspended until other task that is currently in critical section leaves it
by calling mtx_unlock. Mutex protection can by commenting out MUTEX_LOCK. Mutex can also be replaced
with semaphore by commenting out MUTEX_LOCK and uncommenting SEMAPHORE_LOCK.
*/

#include <avr/io.h>
#include "avros.h"
#define MUTEX_LOCK
/* #define SEMAPHORE_LOCK */

int mutex_test_var = 20015;

void mutex_test_task()
{
        /* decrease variable 10000 times */
        int i = 0;
        while (i < 10000)
        {
                i ++;
                #ifdef MUTEX_LOCK
                mtx_lock(0, 0); /* enter critical section */
                #endif
                #ifdef SEMAPHORE_LOCK
                sem_wait(0, 0); /* enter critical section */
                #endif
                mutex_test_var --; /* decrement variable */
                #ifdef MUTEX_LOCK
                mtx_unlock(0);  /* leave critical section */
                #endif
                #ifdef SEMAPHORE_LOCK
                sem_post(0, 1);  /* leave critical section */
                #endif
        }
        exit();                 /* terminate task */
}

void mutex_test()
{
        /* port a is output */
        DDRA = 0xff;
        /* create two tasks running same code */
        unsigned char id1 = create(mutex_test_task);
        unsigned char id2 = create(mutex_test_task);
        /* wait until both tasks are terminated */
        while (signal(id1, SIG_STAT) != RET_NOEXIST ||
               signal(id2, SIG_STAT) != RET_NOEXIST) {}
        /* output variable to port a, should be 15 (00001111) */
        PORTA = mutex_test_var;
        /* sit in loop */
        while(1) {}
}
```

```
/*
This program creates three tasks and synchronizes them using two semaphores. Two of tasks are so
called consumer tasks that wait for new data from third producer task and outputs it to ports a and b.
Both semaphores are initialized to value one when operating system is initialized. When consumer task
calls for sem_wait, value of semaphore is decremented by one. When value of semaphore becomes negative,
task will be suspended until other task (producer) calls for sem_post which increments value of semaphore
and wakes up task that is waiting for semaphore. Producer task stores new values into global variables
data_1 and data_2 and posts to semaphore zero and one. Consumer tasks wait for semaphore and outputs
value of variable each time when producer task posts to corresponding semaphore.
*/

#include <avr/io.h>
#include "avros.h"

unsigned char data_1 = 0;
unsigned char data_2 = 0;

void semaphore_consumer_1()
{
        while(1)
        {
                sem_wait(0, 0);          /* wait for semaphore 0 */
                PORTA = data_1;          /* output data to port a */
        }
}

void semaphore_consumer_2()
{
        while(1)
        {
                sem_wait(1, 0);          /* wait for semaphore 1 */
                PORTB = data_2;          /* output data to port b */
        }
}

void semaphore_producer()
{
        while(1)
        {
                data_1 ++;           /* prepare data for consumer 1  */
                sem_post(0, 1);      /* post to semaphore 0 (increase it's value) */
                sleep(100);          /* sleep for 100 scheduler ticks */
                data_2 += 4;         /* prepare data for consumer 2 */
                sem_post(1, 1);      /* post to semaphore 1 (increase it's value) */
                sleep(100);          /* sleep for 100 scheduler ticks */
        }
}

void semaphore_test()
{
        /* ports a and b are outputs */
        DDRA = 0xff;
        DDRB = 0xff;
        /* create consumer task 1. consumer waits for data from producer by waiting for semaphore. */
        create(semaphore_consumer_1);
        /* create consumer task 2. consumer waits for data from producer by waiting for semaphore. */
        create(semaphore_consumer_2);
        /* create producer task. producer produces data and signals consumer by posting to semaphore. */
        create(semaphore_producer);
        /* sit in loop */
        while(1) {}
}
```

```
flash: program.hex
	avrdude -c jtag1 -b 115200 -p m32 -P /dev/ttyUSB0 -U flash:w:program.hex

program.hex: program.elf
	avr-objcopy -j .text -j .data -O ihex program.elf program.hex

program.elf: avros.o testapp.o signal_test.o mutex_test.o semaphore_test.o lcd.o
	avr-gcc -Wl,-Map,program.map -Wa,--gstabs -mmcu=atmega32 avros.o testapp.o \
	signal_test.o mutex_test.o semaphore_test.o lcd.o -o program.elf

signal_test.o: signal_test.c avros.h avros_asm.h
	avr-gcc -I/usr/lib/avr/include -Wall -g -mmcu=atmega32 -c signal_test.c

mutex_test.o: mutex_test.c avros.h avros_asm.h
	avr-gcc -I/usr/lib/avr/include -Wall -g -mmcu=atmega32 -c mutex_test.c

semaphore_test.o: semaphore_test.c avros.h avros_asm.h
	avr-gcc -I/usr/lib/avr/include -Wall -g -mmcu=atmega32 -c semaphore_test.c

testapp.o: testapp.c avros.h avros_asm.h
	avr-gcc -I/usr/lib/avr/include -Wall -g -mmcu=atmega32 -c testapp.c

avros.o: avros.S avros.h avros_asm.h
	avr-gcc -I/usr/lib/avr/include -Wa,--gstabs -mmcu=atmega32 -a -c avros.S

lcd.o: lcd.c lcd.h
	avr-gcc -I/usr/lib/avr/include -Wall -std=c99 -O2 -DF_CPU=16000000 -mmcu=atmega32 -c lcd.c

clean:
	rm *o *hex *elf

debug:
	avarice --part atmega32 --jtag /dev/ttyUSB0 :4242

disasm:
	avr-objdump -S -D program.elf > dasm

rfuses:
	avrdude -c jtag1 -p m32 -P /dev/ttyUSB0 -U lfuse:r:lfuse:b -U hfuse:r:hfuse:b
	cat lfuse
	cat hfuse
wfuses:
	avrdude -c jtag1 -p m32 -P /dev/ttyUSB0 -U lfuse:w:0xff:m
```